Mémoire de MASTER RECHERCHE

présenté par

Cédric Augonnet

Vers des supports d'exécution capables d'exploiter les machines multicœurs hétérogènes

Encadrement : Raymond Namyst

Février – Juin 2008

Laboratoire Bordelais de Recherche en Informatique (LaBRI)
INRIA Bordeaux
Université Bordeaux 1

Table des matières

Remerciements

Tout d'abord, je tiens à remercier Raymond NAMYST de m'avoir offert la possibilité de travailler sur un sujet passionnant, en me proposant à la fois des suggestions éclairées et une très grande liberté dans la façon d'aborder ce problème. Un grand merci à tous les membres de l'équipe RUNTIME pour leurs précieux conseils, notamment sur l'élevage de poneys dans un Algeco, ainsi que leur accueil toujours aussi chaleureux. Une pensée particulière pour Pierre-André WACRENIER qui, non content de mettre en évidence la quintessence de la littérature germanique, n'a eu de cesse de m'aider par son recul idoine. Enfin, merci à tous ceux qui ont pris de leur précieux temps pour m'aider à améliorer ce document, que ce soit mon tuteur François PELLEGRINI, ou mes divers relecteurs, notamment Élisabeth BRUNET, Cécile ROMO-GLINOS et surtout François TRAHAY.

" Cool cool ! "
Proverbe populaire marcquois

Chapitre 1

Introduction

Alimentée par l'augmentation des capacités de calcul, la demande pour des machines toujours plus puissantes est perpétuelle pour des domaines aussi variés que la sismologie, l'imagerie médicale ou la finance. Au-delà des progrès connus par les techniques de gravure, des limites physiques forcent les architectes à s'orienter vers le parallélisme. Les machines parallèles sont donc omniprésentes en calcul haute performance, aussi bien sous la forme de super-calculateurs ou de grappes de PC qu'à travers les machines multi-processeurs qui sont aujourd'hui tout à fait standard.

Émergence des architectures multicœurs hétérogènes

Cependant, la solution consistant à placer plusieurs processeurs identiques (SMP) sur une même carte mère ne passe pas suffisamment à l'échelle pour maintenir indéfiniment la croissance exponentielle dictée par la loi de MOORE, ce qui se traduit par l'introduction des architectures multicœurs dans les années 2000. Ces architectures multicœurs consistent à bénéficier de l'espace libéré par les gains des techniques de gravure pour mettre plusieurs processeurs, appelés cœurs, par puce. Si cette solution permet d'économiser du silicium, elle permet aussi de s'attaquer à l'épineux problème de la *barrière de la mémoire*. Car l'évolution des performances de la mémoire n'a pas subi une progression aussi soutenue que celle des processeurs, ce qui se traduit par une augmentation inquiétante du nombre de cycles d'horloge nécessaires pour accéder à la mémoire. La conception des architectures multicœurs se veut donc de plus en plus hiérarchiques, avec le partage de ressources matérielles telles que des caches.

En calcul haute performance, il est usuel de chercher la puissance de calcul où qu'elle soit, ainsi l'utilisation de coprocesseurs est devenue courante pour résoudre des besoins très spécifiques. Certaines ressources sont même parfois détournées de leur utilisation initiale pour des besoins de calcul : les processeurs graphiques étant particulièrement performants dans le domaine du calcul vectoriel pour lequel ils sont conçus, leur utilisation comme coprocesseur permet d'obtenir des performances bien au-delà de celles des processeurs récents, pour des classes d'applications adaptées. La force de ces *accélérateurs* réside généralement en la simplicité de leur conception. L'espace disponible sur les puces peut donc être utilisé par y mettre des coprocesseurs simples, parfois nombreux, marquant l'arrivée des machines multicœurs hétérogènes.

Absence de modèle de programmation

Bien que le calcul scientifique bénéficie donc de machines toujours plus puissantes, cette puissance vient au prix d'une complexité de programmation accrue, et le gouffre entre les

performances théoriques et les performances réellement observées ne cesse d'exploser. Le programmeur n'étant pas nécessairement un expert en architecture, et face à des problématiques de portabilité, une aide à la programmation est désormais indispensable. Cette aide peut avoir lieu à divers niveaux, aussi bien avec des langages proposant un support pour le parallélisme que par un réel support au sein du système d'exploitation. Alors que la programmation parallèle avec MPI est maintenant assez courante, ce modèle est très limité dans le cadre des machines multicœurs dont on a encore le plus grand mal à exploiter le potentiel. Et malgré l'émergence de solutions telles qu'OPENMP, on ne dispose encore d'aucun modèle de programmation standard pour le multicœur dans un contexte hétérogène. Étant donnée la grande inertie qui caractérise les codes industriels, ce manque risque d'accroître la sous-utilisation des machines de manière considérable dans les années à venir.

Objectifs

Afin de déterminer le rôle de ce travail, il faut tout d'abord définir à qui il s'adresse. Une abstraction de haut niveau permet généralement d'aider la programmation, alors qu'un modèle unifié répond à des attentes de portabilité, et permet la mise en œuvre de stratégies d'optimisation élaborées.

Deux domaines nous semblent nécessiter une attention particulière. D'une part, les environnements de programmation et les compilateurs (tels que HMPP ou RAPIDMIND) peuvent se reposer sur un support exécutif se chargeant d'ordonnancer les différentes tâches, laissant le compilateur se consacrer à la génération de code efficace. D'autre part, il est courant de recourir à l'utilisation de bibliothèques spécialisées (et optimisées) telles que les implémentations de l'interface BLAS pour l'algèbre linéaire, ou par exemple la bibliothèque PASTIX pour résoudre des grands systèmes linéaires creux. Si les développeurs de telles bibliothèques ont un besoin réel d'un support pour les machines hétérogènes, il est également *primordial* de leur offrir une abstraction qui leur permette d'exprimer les informations dont ils disposent pour aider l'ordonnancement dans le cadre du calcul haute performance.

De manière plus précise, nous allons tout d'abord mettre en œuvre un prototype de support exécutif pour exploiter des machines hétérogène, il s'agira de résoudre les problèmes purement opérationnels en offrant une interface simple qui masque les détails techniques. Ce travail s'articulera autour de deux aspects distincts : la gestion des mouvements de données par le biais d'une de bibliothèque de gestion de la mémoire sur des plateformes hétérogènes, et la mise en œuvre d'une plateforme pour ordonnancer des tâches. Contrairement à la majorité des autres approches, nous allons montrer qu'il est possible d'exploiter des machines réellement hétérogènes, ne laissant par exemple pas un processeur multicœur de coté même si on dispose d'une carte graphique *a priori* plus rapide.

À l'aide de ces prototypes, et disposant des mécanismes de base pour masquer les aspects purement opérationnels de notre support exécutif, nous allons montrer l'importance d'un ordonnancement évolué pour exploiter les machines multicœurs hétérogènes de manière productive.

Chapitre 2

État de l'art

2.1 Architecture : du multicœur à l'hétérogène

Si le parallélisme semble s'être imposé pour maintenir l'évolution de la puissance de calcul, les architectes adoptent désormais une approche hiérarchique, parfois couplée à l'utilisation d'accélérateurs matériels. La puissance de calcul vient donc au prix de machines qu'on ne sait pas encore programmer efficacement.

2.1.1 Adoption du multicœur

Alors que les améliorations des techniques de gravure permettent une réduction de la surface de silicium nécessaire pour graver une puce, cette seule technique ne permet plus d'augmenter les fréquences de processeurs monocœurs sans se heurter à des limites thermiques et électriques.

Limites du SMP

L'approche SMP (*Symmetric MultiProcessing*) qui consiste à ajouter plusieurs puces sur une même carte mère permet de vérifier la loi de MOORE, tout en réduisant les coûts par rapport à des machines mono-processeurs grâce à une meilleure intégration. Cependant, les machines SMP posent un problème de scalabilité puisqu'il est difficile de multiplier le nombre de processeurs accédant à l'unique bus mémoire sans rencontrer un sérieux problème de contention. Bien que les machines NUMA (*Non Uniform Memory Access*) permettent de limiter cette contention, les latences des accès mémoire n'ont pas subi une évolution aussi rapide que celle des fréquences des processeurs. Ce problème connu sous le nom de *Memory Wall* [1] montre qu'il est indispensable d'adopter une approche encore plus hiérarchique pour amortir le coût d'accès à la mémoire en exploitant le principe de localité.

Machines hiérarchiques

Les concepteurs de processeurs doivent donc faire face à 3 limites fondamentales, le *Frequency Wall*, le *Power Wall* et le *Memory Wall* auxquelles l'approche très hiérarchique du multicœur offre une réponse intéressante.

Les architectures multicœurs se caractérisent par l'utilisation de plusieurs niveaux de caches, et surtout de plusieurs cœurs par puce [2]. Le premier bicœur est ainsi apparu dans le POWER4 d'IBM en 2001. Cette période marque aussi le retour des processeurs dits *multithreadés*, avec un support matériel pour plusieurs contextes de calcul partageant des unités de

calcul. Ainsi le PENTIUM IV a introduit l'*hyper-threading* (qui n'est autre que l'implémentation du SMT par INTEL)

Le cas de l'*hyperthreading* est significatif : bien que le gain de performance réel ne se soit pas révélé satisfaisant, IBM persiste à proposer un POWER6 *multi-threadé*. Cet exemple caractérise l'échec des architectes dont les processeurs nécessitent des techniques de programmation bien au-delà de ce que les programmeurs savent faire. L'écart entre performances théoriques et performances observées ne cesse donc de se creuser. Malgré cette difficulté croissante,SUN combine toutes ces approches dans le NIAGARA 2 dont les 8 cœurs comportent chacun 8 contextes matériels : la course aux performances brutes se fait au détriment du programmeur.

Avec l'augmentation du nombre de cœurs et la hiérarchisation de la mémoire, il devient de plus en plus difficile de maintenir une mémoire cohérente sur l'ensemble de la machine. Et si l'ALTIX 4700 de SGI peut aujourd'hui comporter jusqu'à 4096 cœurs avec une mémoire cohérente, le programmeur ne peut plus négliger les effets NUMA. En l'absence d'un protocole de cohérence de cache passant à l'échelle, une solution consiste à renoncer à une mémoire globalement cohérence, et s'en remettre au programmeur pour assurer la cohérence des données. Par exemple, les super-calculateurs tels que le BLUEGENE ne disposent pas de mémoire cohérente entre les nœuds. À l'échelle de la puce, cette stratégie se traduit par l'introduction de multicœurs non cache-cohérents tels que le processeur CELL en 2005 [3]. Là encore, l'évolution des processeurs se fait en marge de ce que les programmeurs sont capables de faire aujourd'hui.

2.1.2 Utilisation d'accélérateurs

Les demandes du calcul haute performance se traduisent souvent par l'utilisation de solutions matérielles pour résoudre les problèmes particulièrement critiques : l'omniprescence du calcul flottant a par exemple conduit à l'introduction de coprocesseurs arithmétiques (FPU). Mais si ces solutions matérielles viennent généralement d'un besoin particulier, à l'image des cartes graphiques qui sont désormais primordiales en visualisation, il s'agit aussi de bénéficier de l'opportunité de trouver la puissance de calcul au meilleur rapport qualité-prix, que ce soit du point de vue de la puissance en tant que telle, ou même du point de vue économique voire énergétique.

Les FPGAs (*Field-Programmable Gate Array*) inventés par XILINX dès 1984 offrent ainsi la possibilité de construire un circuit spécialisé pour un problème donné avec un coût faible comparé à la conception d'une puce. Cette utilisation d'*accélérateurs* performants et bon marché s'explique d'une part par des besoins très spécialisés ; et d'autre part comme une réaction à la demande de croissance perpétuelle qui caractérise le calcul haute performance. Dans l'attente de solutions conventionnelles plus performantes, on observe la multiplication des cartes dites *accélératrices* telles que les coprocesseurs arithmétiques CLEARSPEED, les cartes MERCURY à base de CELL ou les nombreuses cartes dédiées à la simulation physique.

Dans ce marché très dynamique, les solutions à base de cartes graphiques (dont AMD et surtout NVIDIA sont les principaux acteurs) sont aujourd'hui particulièrement utilisées pour des calculs massivement parallèles, bien que ce ne soit finalement qu'une (nouvelle) résurgence des architectures vectorielles apparues dans les années 70 avec les CRAY.

Les possibilités de couplage de code, ou de déport de sections particulièrement critiques expliquent que le nombre de grappes et de supercalculateurs dotés d'accélérateurs soit en très forte croissance comme le montre la Figure 2.1 : SGI offre par exemple la possibilité d'avoir des FPGAs en lieu et place de nœuds de calcul.

Machine	Constructeur	Processeurs	Accélérateurs
TSUBANE (2006)	NEC/SUN	11088×2 (AMD)	360 (CLEARSPEED CSX600)
ROADRUNNER(2008)	IBM	7000×2 (AMD)	13000 (IBM POWERXCELL 8I)
CCRT(2009)	BULL	1068×8 (INTEL)	48×512 (NVIDIA)

FIG. 2.1: Arrivée des accélérateurs dans les machines du TOP500

2.1.3 Vers un multicœur hétérogène

En réponse au dynamisme qui entoure les machines hybrides, on observe deux phénomènes particuliers : un mouvement des accélérateurs vers les fonctionnalités des processeurs modernes, et à terme, une intégration des fonctionnalités des accélérateurs au sein de processeurs multicœurs.

Vers des accélérateurs génériques

Avec le fort engouement autour des technologies d'accélération, les constructeurs souhaitent élargir le spectre de leur marché en offrant des solutions qui se veulent aussi génériques que possible. Ainsi les cartes NVIDIA et les coprocesseurs du CELL disposent maintenant de la double précision, souvent indispensable pour conquérir le marché du calcul scientifique. Il faut cependant noter que ces évolutions architecturales se font en marge des standards [1], ce qui met en avant l'ambiguïté de cette volonté de faire des accélérateurs génériques sans pour autant avoir les capacités des processeurs modernes.

Intégration dans les processeurs

Si les accélérateurs sont une solution de choix pour un vaste spectre d'applications, les processeurs génériques ont souvent vocation à intégrer les fonctionnalités autrefois déportées sur des coprocesseurs puisque l'on peut économiser suffisamment de place sur la puce avec une gravure plus fine. L'exemple des coprocesseurs arithmétiques est significatif puisque tous les processeurs intègrent désormais des fonctionnalités des FPU directement sur la puce, et que l'on trouve également des instructions spécialisées pour les applications de cryptographie.

D'ailleurs, le processeur CELL consiste en une puce hétérogène, où des coprocesseurs particulièrement adaptés aux applications de multimédia collaborent avec un cœur conventionnel de type POWER. Si cette architecture n'est pas nécessairement adaptée à toutes les applications, elle laisse entrevoir l'intérêt d'architectures multicœurs où les cœurs ne sont pas nécessairement homogènes : cette hypothèse est corroborée par les prototypes de TERA-SCALE construits par INTEL, où certains des 80 cœurs peuvent être spécialisés, voire reconfigurables, pour certains types de tâches [4].

De même, le rachat d'ATI par AMD et les effets d'annonces autour du LARRABEE d'INTEL (qui doit intégrer des fonctionnalités proches d'une carte graphique au cœur de la puce) laissent entrevoir la généralisation des processeurs multicœurs hétérogènes.

2.2 Programmation multicœur homogène

Si les machines multicœurs sont désormais majoritaires, les programmer de manière efficace reste encore un véritable problème. On peut manipuler le parallélisme explicitement, ou

[1] Notamment IEEE 754 sur la représentation des virgules flottantes.

considérer des langages offrant directement un support pour le parallélisme.

2.2.1 Gestion explicite du parallélisme

Bien que standard, l'utilisation de l'interface PTHREAD nécessite généralement une très bonne connaissance de l'architecture sous-jacente. Si cette interface permet d'exploiter directement les ressources matérielles, cela vient au prix d'un effort de programmation important, puisqu'il est de la responsabilité du programmeur d'explicitement prendre en compte les problèmes de concurrence directement liés au parallélisme. Dans la plupart des cas, c'est encore la seule approche qui permet d'obtenir de bonnes performances.

Avec l'essor des *grappes* de calcul, l'interface de communication MPI est devenue un standard très important en programmation parallèle. Si cette solution ne permet vraisemblablement pas d'exploiter les avantages d'une approche hiérarchique, les nombreuses implémentation de MPI permettent un portage *a priori* immédiat entre des grappes de calcul et des machines multicœurs. L'effort de programmation pour gérer le parallélisme est là encore assez important, mais l'implémentation de MPI permet de masquer, en partie, la complexité de la machine sous-jacente.

Il faut noter qu'à l'image du FORTRAN vis-à-vis du C, l'inertie qui caractérise les codes industriels est généralement très importante : l'utilisation de MPI y étant relativement récente, il est probable que les codes écrits dans les quelques années à venir ne soit pas encore orientés vers des machines multicœurs.

2.2.2 Se reposer sur un langage

L'intrusivité est un facteur déterminant dans le choix d'une approche pour paralléliser un code. L'approche d'OPENMP consiste à annoter le code pour *indiquer* au compilateur comment paralléliser le code. La simplicité de cette approche semble favoriser l'adoption d'OPENMP, qui permet d'exploiter les machines multicœurs de manière relativement efficace sans pour autant gérer explicitement les problèmes de concurrence.

Une approche prometteuse consiste à combiner OPENMP avec un modèle proposant un parallélisme à plus gros grain tel que MPI, ce qui permet d'exploiter des grappes de nœuds multicœurs, sans pour autant dénaturer la structure du code MPI puisqu'il est possible d'annoter spécifiquement les sections critiques [5].

Il est possible d'utiliser les abstractions du langage pour exprimer les caractéristiques du parallélisme. Ainsi INTEL TBB [6] se repose directement sur le langage C++ pour décrire un parallélisme de tâche directement inspiré par CILK [7], qui permet au langage d'effectuer des optimisations complexes, grâces à la structure même des objets ainsi parallélisés.

2.3 Programmation d'accélérateurs

Dans cette section, nous considérons les différentes solutions adoptées pour programmer deux classes d'accélérateurs significatifs : d'une part les processeurs graphiques, et d'autre part l'architecture CELL. Nous ne nous attardons pas sur les solutions de type FPGA, dont l'utilisation semble encore peu adaptée à la résolution de problèmes génériques.

2.3.1 GPGPU

OWENS et al. montrent que les cartes graphiques ont été utilisées pour un traiter un vaste spectre d'applications [8].

Avant de disposer de cartes programmables, la seule technique disponible consistait à traduire les problèmes sous la forme d'opérations sur des *textures* et à détourner les API graphiques standards telles que DIRECTX et OPENGL. Non seulement cette approche nécessite une très bonne connaissance de ces API, mais il est parfois difficile d'adapter le problème à des pipelines graphiques qui ne sont pas prévus pour un tel usage. Ces cartes ne sont par exemple pas conçues pour effectuer des accès mémoires quelconques (eg. *scatter* et *gather*), et la gamme des instructions disponibles sur les processeurs graphiques est restreinte. Les cartes programmables permettent de détourner le mode de fonctionnement des pipelines graphiques, dont l'utilisation peut être contre-productive dès lors que l'on s'éloigne du cadre usuel de la visualisation.

Les langages NVIDIA CG [9], HLSL (pour DIRECT3D) et GLSL (*OpenGL Shading Language*) [10] permettent de programmer ces cartes à l'aide d'une sémantique similaire au C. Cependant, ils ne manipulent que des constructions directement liées au matériel graphique (eg. des *textures*), même si GLSL permet par exemple de manipuler des entiers, qui n'ont pourtant pas d'interprétation matérielle directe. Des langages de plus haut niveau, comme ASHLI [11] ou SH [12] permettent ensuite de générer un code (eg. en HLSL) qui se charge du partitionnement des données et de la mise en œuvre de l'ensemble du pipeline. L'utilisation de primitives graphiques comme *objet de première classe* n'étant pas nécessairement productive, BROOK [13] ajoute des *flux* au langage le C, alors que SCOUT [14] offre des structures de données adaptées à la visualisation scientifique. La bibliothèque GLIFT [15] généralise cette approche et permet d'utiliser des structures de données évoluées.

Face au succès des cartes graphiques programmables, les constructeurs tels que NVIDIA et AMD ont modifié leurs architectures pour y ajouter quelques fonctionnalités importantes pour la démocratisation des GPGPU, comme un contrôle de flot plus élaboré, et une gestion des accès mémoire de type *gather/scatter*. À l'image de la programmation de *shader* où l'on manipule des primitives graphiques à bas niveau, l'interface CTM (*Close To Metal*) d'AMD fourni un ensemble d'instructions matérielles pour contrôler les cartes RADEON récentes [16]. Mais si cette solution de beaucoup trop proche du matériel n'a pas reçu le succès attendu, il n'en est pas de même pour NVIDIA CUDA [17] dont le compilateur manipule un code très proche du C. CUDA propose en effet une abstraction du matériel qui ne nécessite plus d'être un expert en graphisme pour faire le moindre calcul. En réponse, les cartes AMD FIRESTREAM utilisent une abstraction de plus haut niveau CAL (qui fait elle-même appel à CTM) ainsi qu'une version du langage BROOKS directement optimisée pour ce type de cartes [18]. APPLE annonce également l'environnement OPENCL (*Open Compute Library*).

Malgré l'émergence des approches de type CUDA, l'absence d'une solution de remplacement standardisée explique que certains persistent à utiliser les API graphiques pour le calcul dans un souci de portabilité [19]. Mais il se dégage que l'utilisation de carte graphique se développe principalement par la possibilité d'utiliser des abstractions de haut niveau bien plus accessibles.

2.3.2 Cell

La conception très particulière du CELL, notamment la nécessité d'effectuer les mouvements de données à l'aide de mécanismes explicites, fait qu'il est encore très difficile de fournir une interface de programmation efficace de haut niveau. Cette difficulté se traduit généralement par l'utilisation de la bibliothèque LIBSPE, qui est le pendant de l'interface PTHREAD pour le contrôle des coprocesseurs. Cette approche de très bas niveau offre un contrôle *total*, mais nécessite une certaine expertise dans plusieurs domaines de compétence [20]. D'une part il est indispensable de comprendre les détails de l'architecture du CELL, et d'autre part, il est indis-

pensable de recourir à un modèle de programmation *asynchrone* pour exploiter le potentiel du CELL [21].

Un modèle d'exécution s'est cependant démarqué : l'application soumet des tâches avec une interface simple, qui les distribue sur les coprocesseurs ; chaque coprocesseur mettant en œuvre un automate qui exécute un pipeline de tâches. Ce modèle est par exemple adopté par l'interface IBM ALF [22], le langage CHARM++ [23], le support exécutif GORDON[2] ou encore le MERCURY MCF [24]. Toutes ces approches se distinguent par une gestion de la mémoire plus ou moins souple, et par la mise en œuvre de stratégies particulières pour exécuter le pipeline de tâches. D'autres modèles d'exécutions existent, ainsi les coprocesseurs du CELL sont parfois utilisés en pipeline, dont chaque SPE constitue une étage [25].

Mais s'il est encore très difficile d'écrire du code efficace malgré l'aide de tels supports exécutifs, il est autrement plus complexe de générer un code qui exploite le CELL de manière transparente. Cela se traduit sur les performances de projets plus ambitieux, tels que CELLSS qui permet d'écrire du code OPENMP, en laissant le langage se charger du déport de calcul [26]. Mais à terme, il est probable qu'une telle démarche permette une démocratisation du calcul sur CELL, à l'image de l'essor qu'a connu le calcul sur carte graphique avec CUDA.

2.4 Programmation hybride

Avec la multiplication des technologies d'accélérateurs et des processeurs multicœurs, il semble peu productif de porter une application sur chacune des architectures disponibles. Il est également regrettable qu'un programme n'exploite qu'un accélérateur donné alors que la machine dispose d'autres ressources de calcul, notamment d'un processeur multicœurs. Dans cette optique de portabilité, et pour bénéficier de l'intégralité des ressources d'une machine, il est donc fondamental de développer des modèles hybrides, permettant par exemple de répartir un calcul simultanément sur les cœurs des processeurs *et* sur les différentes cartes graphiques. Les architectures multicœurs restent donc aussi des cibles de choix pour ces langages hybrides.

Le défi le plus important consiste alors à permettre une exécution sur des machines réellement hétérogènes, plutôt que la seule portabilité qu'autorise une compilation vers différentes cibles. Selon le type d'objets manipulés, deux approches se distinguent. Suivant un *parallélisme de données*, il est possible de considérer les tableaux ou les matrices comme des objets de première classe, comme le font RAPIDMIND [27], BROOKS [13], PEAKSTREAM, STREAMIT ou MAPREDUCE. Il est par contre possible de s'orienter vers un *parallélisme de tâches*, en offrant des abstractions déporter des fonctions de manière plus ou moins explicite à l'image des projets MERGE [28], MATCH [29] ou SEQUOIA [30]. Enfin, il est possible d'automatiser la génération de code pour accélérateurs en utilisant des environnements de compilations comme EXOCHI [28], R-STREAM [31] ou encore HMPP [32]. Mais l'absence de standard établi rend cette tâche bien difficile, puisque chaque constructeur développe son propre modèle dans l'idée d'en faire un standard *de facto* (généralement celui qui demande le moins d'effort au constructeur avec un modèle qui s'appuie directement de l'implémentation matérielle sous-jacente). Ainsi IBM essaye de généraliser ALF comme une plateforme multicœur qui ne se limite pas au CELL ; NVIDIA pousse CUDA comme un modèle de programmation parallèle générique comme le montre l'initiative de MCUDA [33]. À l'inverse, les efforts sur OPENMP [26] et MPI [34, 35] démontrent que l'on peut envisager d'utiliser des approches reconnues, bien que cela nécessite un effort de standardisation indispensable.

[2]Nous concevons GORDON dans le cadre d'une collaboration personnelle avec MAIK NIJHUIS, HERBERT BOS et HENRI BAL à *l'Université Libre d'Amsterdam* afin d'offrir un support efficace du processeur CELL dans le projet SCALP.

Chapitre 3

Propositions

3.1 Problématique

Avec l'émergence bien réelle des architectures hybrides, des problématiques apparaissent à tous les niveaux de la chaîne de programmation. Nous avons vu qu'il n'y a pas encore de langage ou de modèle de programmation standard pour unifier les différents types d'accélérateurs.

3.1.1 Objectifs de ce travail

Du travail nécessaire à tous les niveaux

Si la tendance est à un parallélisme explicite, les compilateurs doivent encore fournir des efforts importants. Il leur faut par exemple distinguer la génération de code pour un cœur de celle pour CUDA. À plus bas niveau, les constructeurs continuent de concevoir de nouveaux accélérateurs en permanence, ce qui accentue le besoin de mettre en œuvre un support efficace au sein des systèmes d'exploitation pour combler l'écart qui se creuse entre la complexité des architectures et les capacités réelles des programmeurs.

Enfin, et c'est sur cet aspect que notre travail se porte, il faut mettre en place des supports exécutifs pour disposer d'abstractions de haut niveau nécessaires à une programmation efficace sans pour autant se soucier de détails purement techniques liés à l'interaction avec le système et à la variété des approches proposées par les constructeurs. Au delà du besoin de fournir une abstraction des ressources, les supports exécutifs doivent être capables d'ordonnancer et de répartir les tâches de calcul en prenant en considération les avantages ainsi que les limitations des différents accélérateurs. Il est particulièrement important que l'ordonnancement tire parti des informations fournies par les compteurs de performance, ce qui est difficile sans l'aide des outils ou des fonctionnalités typiquement offerts par un support exécutif.

Une aide pour le programmeur

Offrir une abstraction de haut niveau permet un certain bénéfice du point de vue de la programmabilité. Plus que simplifier la programmation, un environnement de programmation qui se charge d'exécuter les noyaux de calcul de manière transparentes permet la cohabitation des plusieurs accélérateurs. Ces accélérateurs peuvent eux-mêmes être homogènes (eg. plusieurs cartes graphiques identiques), ou hétérogènes (eg. une carte graphique et un CELL).

Il faut déplacer la complexité de programmation autant que possible. Si l'écriture des noyaux des calculs reste à la charge du programmeurs ou du compilateur, nous proposons

d'automatiser les interactions entre les tâches ainsi que les mouvements de données sous-jacents (bien que la *beauté* de l'interface reste un objectif secondaire). Notre solution vise à permettre des techniques et des politiques d'ordonnancement évoluées qui sont elles particulièrement difficiles à mettre en œuvre. Il s'agit donc de laisser le programmeur se concentrer sur l'écriture de noyaux de calcul efficaces tout en lui offrant des moyens expressifs pour décrire son algorithme parallèle plutôt que s'attarder sur les différents détails techniques.

Identification des défis à relever

Pour comprendre quelles sont les difficultés pour mettre en œuvre un support exécutif, il faut mettre en évidence certains problèmes particulièrement difficiles.

Si on peut accélérer un calcul en le déportant, il arrive aussi que ce déport soit plus coûteux que le calcul lui-même. Il faut donc prendre garde à n'utiliser les accélérateurs que pour des opérations avec une très forte densité de calcul, et dont la granularité est suffisante. Le support exécutif peut alors indiquer si l'exécution d'une tâche a été efficace, ce qui donne des indications au programmeur qui peut adapter la granularité en conséquence.

Ce problème est réputé pour être difficile puisque déterminer la granularité optimale requiert généralement une certaine expertise et beaucoup d'expérimentation. Une solution classique consiste à effectuer des tests qui vont déterminer les paramètres *optimaux* de manière statique. La bibliothèque ATLAS recherche les paramètres optimaux à l'installation [36], alors que la bibliothèque PASTIX utilise un pré-calcul pour un placer les calculs statiquement. Une autre approche consiste à effectuer des tests pour étudier le comportement de la machine afin générer un code adapté à l'architecture [21]. Si il est déjà compliqué de choisir une granularité adaptée pour une architecture donnée, cela devient particulièrement complexe pour une machine hétérogène où une approche purement statique n'est plus suffisante.

Fournir suffisamment de parallélisme est un problème critique pour exploiter les accélérateurs qui ont tendances à être massivement parallèles, voire vectoriels. Il faut donc une articulation entre les modèles de programmation qui extraient le parallélisme, et les modèles d'exécution qui se chargent de fournir suffisamment de travail là où cela se révèle le plus rentable.

3.1.2 Critères de conception d'un support exécutif

Nous avons tout d'abord isolé les composants essentiels pour mettre en œuvre notre support exécutif. Trois éléments sont alors distingués :

Modéliser la machine hétérogène. Une représentation structurée de la machine est cruciale pour exploiter le potentiel des accélérateurs de calcul : ainsi on évitera de placer une tâche de calcul très sensible aux effets de cache (eg. un appel à la bibliothèque BLAS) sur un processeur qui est également utilisé pour contrôler un accélérateur, qui serait susceptible d'effectuer des opérations ayant des effets de cache potentiellement très néfastes. Des effets de type NUMA [37] ou NUIOA [38][1] peuvent également être pris en compte si on modélise la machine parallèle avec soin.

Structuration en tâches. Nous proposons d'utiliser une structure de donnée, appelée *codelet*[2], qui permet de représenter les informations nécessaires au déport de tâches de calcul ainsi que les dépendances qui peuvent exister entre ces différentes tâches.

[1] Il est par exemple préférable de placer un *thread* faisant des communications sur un cœur proche de la carte réseau.

[2] Cette dénomination apparaît dans la terminologie de HMPP.

Gestion des mouvements de données. Dans le contexte d'une machine hétérogène, les mouvements de données peuvent s'avérer particulièrement délicats à mettre en œuvre du fait de l'absence de mémoire partagée de manière cohérente. Dès lors que ces transferts de données sont particulièrement critiques pour les performances, il faut aider le programmeur à maintenir des données cohérentes au sein d'une machine hétérogène.

3.1.3 Vers des politiques d'ordonnancement avancées

Avec le support du matériel, et les indications du programmeur, il est possible d'appliquer des optimisations sur la stratégie d'ordonnancement des *codelets*.

Une approche portable

Évoquer la portabilité dans le contexte de la programmation de machines hétérogènes peut sembler utopique. En effet on ne dispose pas d'un modèle de programmation commun à tous les accélérateurs, et le portage d'une application sur une nouvelle architecture demande généralement un domaine de compétence particulier. Le principal frein à l'évolution des environnements de programmation hybride est donc l'absence d'une interface standard reconnue. Dans cette optique, procéder dans un souci de portabilité est un aspect crucial pour concevoir un support exécutif viable, utilisable par un programmeur qui n'est pas un expert des différentes architectures d'accélérateur.

Supporter une nouvelle architecture d'accélérateur doit se faire avec un effort de programmation raisonnable, sans pour autant remettre en cause l'architecture du cœur de notre système. Nous adoptons une architecture modulaire qui permet de contrôler les ressources de calcul avec souplesse.

La structure des algorithmes ne doit pas être affectée par l'ajout ou le retrait d'un type d'accélérateurs, et l'utilisation de plusieurs accélérateurs identiques doit être transparente.

Un moteur d'ordonnancement générique

L'approche de la plateforme BUBBLESCHED [39] est intéressante de ce point de vue : dès lors que l'on dispose de **mécanismes** pour mettre en œuvre l'ordonnancement (que ce soit pour des *threads* dans le cas de MARCEL ou de *codelets* en ce qui nous concerne), il est possible de concevoir des **politiques** d'ordonnancement qui utilisent ces mécanismes. Cette idée est d'ailleurs confortée par la bibliothèque de communication NEWMADELEINE qui applique des mécanismes de haut niveau pour mettre en œuvre des schémas de communication évolués [40].

Ces travaux font également apparaître la nécessité d'une interface de programmation simple mais suffisamment expressive pour permettre au programmeur de fournir des indications au moteur d'ordonnancement pour exploiter le potentiel de ces politiques. L'absence *actuelle* de standard nous laisse une très grande liberté de ce point de vue.

Enfin, et c'est là un point important, les résultats obtenus par ces plateformes mettent en évidence l'absence de l'existence d'une politique *ultime* adaptée à tous les problèmes. Et il est probable que si une politique est optimale pour une classe de problèmes, elle ne sera pas adaptée à d'autres applications. Ceci implique donc la nécessité de faire cohabiter diverses politiques, le programmeur ayant alors la possibilité de choisir la plus adaptée parmi celles à sa disposition. Il est possible d'assister le moteur d'ordonnancement à l'aide de divers informations : on peut utiliser les compteurs matériels pour analyser le comportement réel de l'application, et il est possible de prendre en compte les indications des programmeurs.

Des informations matérielles exploitables

Il existe de nombreux indicateurs qui permettent à un programme de connaître son propre comportement du point de vue du matériel sous-jacent. La plupart des architectures modernes disposent en effet de compteurs matériels qui mesures de nombreux paramètres (eg. l'utilisation du cache, la densité de calcul ...).

Malheureusement il est souvent inconcevable d'écrire du code portable qui exploite ces compteurs. Et si certains environnements les utilisent, ce sera le plus souvent pour une analyse *post-mortem* afin de mettre en évidence les parties du code qu'il faut optimiser. Plusieurs facteurs expliquent cette réticence à recourir aux compteurs matériels : leur accès peut être particulièrement technique et nécessiter l'utilisation d'une bibliothèque telle que PAPI. L'aspect hétérogène ne fait d'ailleurs qu'accroître cette difficulté.

Un exemple de politique pourrait être de favoriser l'exécution de tâches ayant exhibé une forte utilisation des instructions vectorielles sur des accélérateurs particulièrement efficaces pour ce type de parallélisme.

Des directives du programmeur

Si le matériel est susceptible de fournir des informations, le programmeur est généralement d'autant plus à même de donner des indications pour guider le support exécutif.

Un des défis soulevés par la Section 3.1.1 consiste à sélectionner une granularité adaptée. Une stratégie d'ordonnancement évoluée peut apporter des éléments de réponse à ce problème : si le programmeur fourni un modèle de coût pour prévoir le temps d'exécution des différentes tâches, on peut alors déterminer s'il existe un meilleur ordonnancement pour des tâches de granularité différentes. Cet exemple montre donc que l'ordonnancement bénéficie d'une aide explicite du programmeur. Il faut donc que notre interface de programmation permette de transmettre les suggestions du programmeur au moteur d'ordonnancement.

3.2 Structurer les traitements : la notion de codelet

Afin d'exprimer la structure de tâche ainsi que de représenter les dépendances entre les tâches qui constituent l'application, nous proposons d'utiliser la notion de *codelet* dont la Figure 3.2 donne un aperçu.

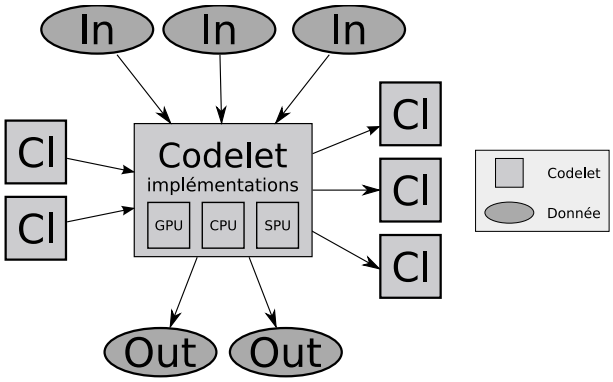


FIG. 3.1: Structurer les dépendances entres tâches et données avec des codelets

3.2.1 Définir un modèle d'exécution

La principale caractéristique des *codelets* est de spécifier quel code exécuter selon l'accélérateur (ou le cœur) qui sera finalement désigné pour exécuter la tâche décrite. Le programmeur peut spécifier qu'une *codelet* est ordonnançable sur un sous-ensemble des ressources de calcul si nécessaire, par exemple lorsqu'il n'y a pas de méthode disponible pour un accélérateur donné, ou quand une tâche est particulièrement contre-indiquée sur tel ou tel accélérateur.

Afin de permettre le déport de calcul, il faut être capable de déplacer les données nécessaires à un calcul. Une *codelet* doit donc contenir la description de toutes les données qui doivent être disponibles lors de son exécution. On peut par exemple décrire les données utilisées à l'aide d'une interface de haut niveau telle que celle décrite dans la Section 3.3. Afin de minimiser les transferts de données, et de permettre diverses optimisations, on précise le mode d'accès pour chacune de ces données (eg. lecture seule).

Il faut définir un modèle d'exécution de ces *codelets*. Contrairement aux appels de fonctions distantes (RPC), l'exécution des *codelets* est asynchrone, et il n'est pas nécessaire spécifier précisément la ressource sur laquelle elle aura lieu. Au contraire, on adopte ici un modèle de programmation par continuations (ou *callback* en anglais). Le programmeur soumet des *codelets* au support exécutif, et ce dernier est alors responsable de l'exécution de la tâche dans les meilleures conditions possibles, puis d'exécuter la continuation.

Dans ce modèle très simple, on suppose donc que le support exécutif a le droit d'appliquer des optimisations plus ou moins agressives tant que l'on assure l'exécution de la tâche puis celle de la continuation. Les optimisations possibles sont très variées, on pourra par exemple imaginer que le support exécutif réordonne les *codelets* ou automatise des techniques de type *multi-buffering* qui permettent le recouvrement de calcul et de communications, souvent indispensable pour exploiter les accélérateurs asynchrones [41].

3.2.2 Vérification des hypothèses initiales

Comme nous l'avons précisé précédemment, ce modèle est principalement ciblé sur les environnements de compilation et sur les bibliothèques spécialisées.

Dans le cas des bibliothèques, le programmeur connaît généralement très bien la structure de son application, et l'intérêt d'un système capable d'optimiser l'exécution en prenant en compte les indications du programmeur est assez clair. Quant à la génération de code pour les divers accélérateurs, le programmeur peut s'en remettre à une solution au niveau de la compilation, ou plus simplement fournir les différentes versions du code optimisés.

Dans le contexte d'une approche au niveau de la compilation, et en supposant que le compilateur soit capable de générer diverses versions du code selon les accélérateurs ciblés, il ne reste qu'à exprimer les relations de dépendances entre les tâches. Ceci est généralement à sa portée si on suppose l'utilisation d'une représentation intermédiaire sous la forme d'un graphe de dépendance avant l'étape de génération de code. Là encore, il est envisageable de disposer d'indications de la part du compilateur ou même du programmeur dans le cas d'un langage augmenté de quelques directives adaptées.

Si le modèle de programmation asynchrone à base de continuations n'est pas adapté, il est possible d'ajouter une autre interface faisant appel au modèle des *codelets* de manière sous-jacente. On peut par exemple utiliser la bibliothèque de *threads* MARCEL avec une interface synchrone, où chaque thread peut soumettre une *codelet* avec une sémantique proche d'un appel de fonction.

3.2.3 Exprimer les dépendances de codelets

La plupart des applications parallèles ne sont pas de simples collections de tâches soumises à l'exécution mais de véritables graphes de tâches à ordonnancer. Il est donc important d'offrir un support efficace pour exprimer les dépendances entre les diverses codelets.

Utiliser les continuations

Une approche simple consiste à implémenter les dépendances en permettant le lancement de *codelets* dans les continuations : si B dépend de A, il suffit de lancer B dans la continuation de A. Une limitation apparaît si le graphe des tâches n'est plus un arbre mais un graphe acyclique orienté (ou DAG). Il faut en effet que le programmeur détecte la terminaison de toutes les *codelets* et qu'il fasse attention à éviter les problèmes de concurrence (qui sont tout à fait possibles puisque l'on peut réordonner les *codelets* et que les continuations peuvent être exécutées de manière concurrente).

Bénéfices d'une vision plus globale de l'algorithme

Un modèle avec un réel support des dépendances permet un gain évident de programmabilité puisqu'on évite une gestion explicite des dépendances par le programmeur. Cela aide aussi l'ordonnancement puisque dans l'absence d'un tel support, l'ordonnanceur est tributaire du programmeur pour extraire suffisamment de parallélisme de l'application. Pour ce faire, il est possible d'associer une étiquette à chaque tâche et de construire un graphe des dépendances entre les étiquettes pour que le support exécutif puisse déterminer quand une tâche peut être ordonnancée sans que le programmeur ne le fasse explicitement.

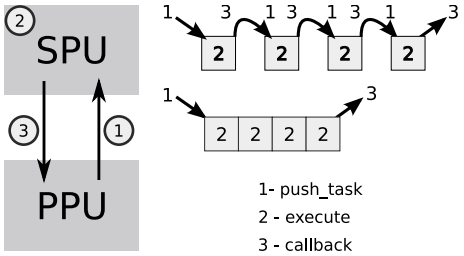


FIG. 3.2: Réduire les interactions entre les cœurs

La Figure 3.2 montre le cas d'une chaîne de tâches à exécuter sur un des coprocesseurs du CELL. Sans support particulier, il faudra lancer un calcul, puis attendre que la continuation soit exécutée sur le processeur principal pour lancer la *codelet* suivante. Or l'exécution d'une continuation est susceptible de nécessiter le réveil d'un *thread* sur le processeur principal, le surcoût entre chaque tâches devient alors non négligeable, non seulement du point de vue du temps d'exécution qui risque d'être directement limité par le temps d'exécution de la continuation, mais on observe une forte dégradation des performances globales du système si on appelle trop de continuations. En revanche, s'il n'est plus nécessaire d'effectuer un va-et-vient entre le coprocesseur et le processeur entre chaque tâches, il est possible d'exécuter toute une chaîne avec un surcoût minime en n'appelant qu'une unique continuation en fin de chaîne.

Le support exécutif peut extraire du parallélisme de manière automatique, ce qui est bénéfique pour l'équilibrage de charge. Il en résulte aussi une diminution du surcoût sur le système grâce à la réduction des interactions entre les accélérateurs et le support exécutif. Des optimisations agressives dans les politiques d'ordonnancement sont possibles : on peut exploiter

l'affinité mémoire lorsqu'une *codelet* réutilise les données d'une autre *codelet* dont elle dépend directement.

3.3 Structurer les données : une bibliothèque de gestion de données

L'absence de mémoire partagée et cohérente est une difficulté majeure de la programmation sur machines hétérogènes. Le programmeur doit donc maintenir des données cohérentes de manière logicielle. Si ce problème est complexe pour un accélérateur donné, il l'est d'autant plus avec plusieurs types d'accélérateurs qui collaborent. Il faut donc un moyen de masquer cette hétérogénéité des techniques pour déplacer des données : plutôt que de demander au programmeur de manipuler le contrôleur DMA asynchrone d'un SPE ou les différentes primitives de copie mémoire synchrones proposées par CUDA, il faut fournir des primitives de haut niveau, telles que la lecture ou l'écriture d'une donnée.

Au delà de l'hétérogénéité qui pose d'évidents problèmes de programmabilité, les transferts mémoires sont particulièrement critiques du point de vue des performances : si une carte graphique NVIDIA peut manipuler de l'ordre de 50Go/s entre sa mémoire locale et les processeurs embarqués, le bus PCI ne peut transférer que 2 ou 3 Go/s, offrant un goulet d'étranglement très important. De même, étant donné le peu de mémoire embarqué sur les SPUs du CELL, il est indispensable de gérer les données de manière efficace si on veut que chaque processeur dispose en permanence de suffisamment de données à traiter pour ne pas gâcher du temps de calcul.

3.3.1 Modèle de cohérence mémoire

Plus qu'un simple support pour déplacer des données, il faut aider le programmeur en automatisant les mécanismes de cohérence mémoire. On doit donc permettre un accès transparent à une donnée de part et d'autre de la machine, quitte à récupérer une copie à jour si nécessaire.

Mécanismes pour maintenir la cohérence

Si chaque nœud de calcul est associé à un nœud mémoire, la gestion de données cohérentes est similaire aux problématiques liées aux architectures dites COMA (*Cache-Only Memory Architectures*) : chaque donnée n'est pas associée à un nœud en particulier, et il est nécessaire de garder au moins une copie de chaque donnée. Et s'il n'y a finalement que peu d'architectures qui ont implémenté ce modèle au début des années 1990s, comme le KSR-1 ou la DDM, le besoin de construire des machines qui passent à l'échelle reste tout à fait d'actualité. Notre approche est similaire à celle du KSR-1 puisque nous relâchons le modèle COMA en supposant qu'il existe toujours un nœud mémoire principal contenant la donnée, même si cette dernière n'est pas à jour.

Nous devons choisir quel type de mécanisme est utilisable pour maintenir la cohérence des données dans notre cas. Les mécanismes de type *snooping cache* doivent être écartés puisque l'on ne dispose pas de support matériel pour diffuser les invalidations (*write-invalidate*) ou les mises à jour (*write-update*). Il reste donc les mécanismes dits *directory-based* où chaque processeur doit être capable de consulter l'état d'une donnée dans une structure accessible par tous pour la maintenir dans un état cohérent. Si la latence pour accéder au répertoire est généralement plus élevée que dans le cas du *snooping*, l'absence de diffusion permet le passage à l'échelle.

Choix d'un modèle mémoire

Afin de réduire autant que possible les transferts mémoire, nous adoptons une approche paresseuse où les mouvements de données ne sont effectués que lorsque c'est parfaitement nécessaire, par exemple si on souhaite lire une donnée alors que celle-ci n'est pas disponible localement. Par opposition au modèle dit *write-through* où chaque écriture se répercute sur un éventuel nœud principal, nous utiliserons donc un modèle de type *write-back*, qui retarde les mouvements de données autant que possible.

La comparaison avec les architectures COMA permet d'éliminer un certain nombre de candidats pour le choix du protocole de cohérence de cache (eg. MSI, MOESI, MESI, ...) . Étant donné qu'il n'y a pas de nœud principal, la distinction entre un état *modifié* et les états *exclusif* ou *owned* perd son intérêt, ce qui nous guide naturellement vers un protocole de MSI, qui a le mérite d'être particulièrement simple à mettre en œuvre.

La figure 3.3 montre l'exemple d'une carte graphique qui fait un accès en *lecture-écriture* sur une donnée qui est disponible à la fois en mémoire principale et sur une autre carte. Cette opération induit un transfert mémoire, et seule la carte graphique dispose de la donnée au final, les autres copies étant invalidées.

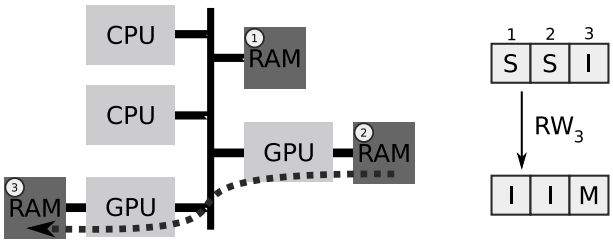


FIG. 3.3: Modèle mémoire avec un protocole MSI

Combiner les modèles mémoires

Dans un souci de généralité, il faut considérer qu'il existe des architectures qui n'autorisent pas l'accès de leur mémoire à une ressource de calcul quelconque (eg. un autre accélérateur). Cela nous pousse donc à reconsidérer l'utilisation systématique d'un modèle *write-back* [3]. Nous avons donc relâché le modèle décrit précédemment : s'il est impossible d'implémenter un mécanisme pour aller récupérer des données distantes (*pull*), on utilise un mécanisme pour déposer les données (*push*).

Ce compromis est particulièrement intéressant pour faire collaborer les différents SPUs d'un CELL. Nous pouvons par exemple utiliser le modèle *write-through* pour répercuter la modification d'une donnée sur un SPU vers un ou plusieurs de ses voisins. En combinant cette fonctionnalité avec une description des dépendances entre les tâches telle que décrite dans la Section 3.2.3, il est possible d'implémenter un pipeline entre les différents SPUs, alors que ce modèle est généralement délicat à mettre en œuvre [25]. Plus généralement, de tels résultats sont extensibles à un modèle dit de *streaming*.

En conséquence, nous adoptons un modèle mémoire relâché pour palier aux limitations des pilotes constructeurs, et dans l'objectif secondaire d'améliorer le support du *streaming*.

[3] Des limitations purement techniques propres à CUDA empêchent par exemple à un *thread* quelconque d'accéder à la mémoire de la carte graphique. Toute modification d'une donnée sur une carte NVIDIA sera donc suivie d'une mise à jour de la copie en mémoire principale (*write-through*).

3.3.2 Une approche hiérarchique de haut niveau

Manipuler des données de haut niveau

Dans la Section 3.3.1, nous avons proposé un modèle pour maintenir des données cohérentes sans préciser comment nous définissons une *donnée*. Afin de rester génériques, et puisque la gestion des mouvements de données et le maintient de leur cohérence peut se faire indépendamment, nous ne pré-supposons pas de représentation particulière. Si une interface telle que celle offerte par les BLAS prédomine pour les problèmes de calcul matriciel, il existe des problèmes pour lesquels elle n'est pas adaptée.

Ainsi il est possible de mettre en œuvre plusieurs représentations des données, et de sélectionner la plus adaptée à un problème, ou même de combiner plusieurs représentations dans un même programme. Ceci n'est possible que puisque nous supposons que les mouvements de données sont orchestrés par le protocole de cohérence, qui ne requiert que des primitives très simples. On peut également proposer une implémentation optimisée de chacune de ces primitives pour diverses interfaces.

Projeter les données selon l'algorithme

Tout comme nous structurons l'application à l'aide de *codelets*, il est indispensable de structurer les données manipulées. En effet, dans le cas d'un algorithme qui fait un calcul sur un tableau de plusieurs millions d'éléments, où chaque sous-tâche travaille sur une sous-partie du tableau, nous ne pouvons pas utiliser une primitive qui effectue des copies mémoires de l'ensemble du tableau à chaque modification de quelques éléments. Ce serait non seulement inefficace, mais dans le cas d'accélérateurs ayant une faible capacité mémoire, il serait impossible de manipuler la donnée que constitue le tableau.

Un aspect important de la programmation de machines hétérogènes est l'utilisation de *paradigme de parallélisme de donnée*, qui se traduit par le besoin de décrire les données manipulées par chaque *codelet*. Or chaque tâche est généralement chargée de traiter un sous-ensemble des données : il est par exemple fréquent de paralléliser des problèmes d'algèbre linéaire avec un découpage par bloc. Nous exploitons cette propriété en mettant en place un mécanisme pour ne manipuler qu'une sous-partie d'une donnée, tout comme on manipule des sous-matrices dans un algorithme par bloc.

La notion de filtre

La représentation des sous-données est un problème au moins aussi difficile que la représentation d'une donnée. Ainsi nous poursuivons notre démarche en n'imposant pas d'interface particulière, grâce à l'utilisation de *filtres*.

L'introduction des *filtres* peut se comprendre par un exemple simple : si on suppose un simple tableau découpé de manière régulière, et que l'on souhaite déterminer dans quel élément de la partition se trouve un indice donné, il faut que le support exécutif effectue un calcul coûteux, alors que cette information est probablement connue du programmeur. L'utilisation de *filtres* permet de ne pas perdre d'information, évite un surcoût important tout en maintenant notre approche de haut niveau : le modèle de cohérence mémoire n'a *pas* à connaître l'organisation concrète de la donnée en mémoire.

Un *filtre* a pour rôle de structurer une donnée sous la forme d'un ensemble ordonné : tout comme nous manipulons une donnée, il est possible de manipuler la $i^{\text{ème}}$ sous-donnée d'une donnée sur laquelle un filtre a été appliqué. Et si nous supposons que l'utilisation des *filtres*

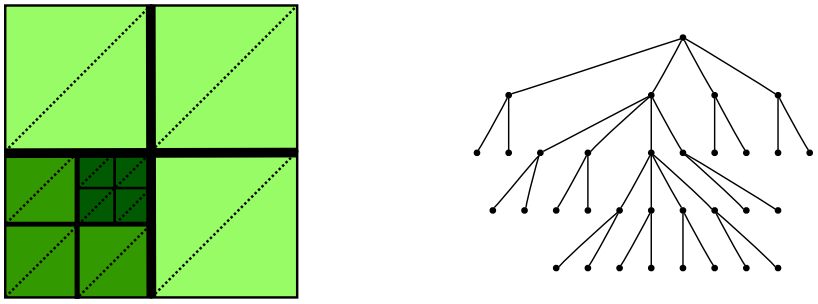


FIG. 3.4: Une donnée hiérarchique et sa représentation arborescente

peut se faire de manière récursive, un découpage par blocs selon les deux dimensions d'une matrice s'exprime par exemple par l'application de deux filtres de découpage par blocs sur la matrice.

Cela signifie que dans notre modèle, chaque sous-donnée devient une donnée à part entière, sur laquelle le modèle de cohérence mémoire doit continuer de s'appliquer. Or si le découpage défini par un *filtre* n'est pas une partition, il est possible que deux données ne soient pas disjointes, ce qui n'est pas possible puisque la cohérence de chaque donnée doit être maintenue de manière indépendante.

La Figure 3.4 donne un exemple de donnée découpée à l'aide de *filtres*. On peut donc combiner différents types de *filtres* puisqu'on découpe ici par blocs puis en triangles.

Une structure arborescente

Nous avons donc un choix important à faire : modifier notre modèle mémoire, ou s'assurer que chaque donnée est indépendante. Modifier le modèle mémoire imposerait d'adopter un protocole qui s'assure de mettre à jour toutes les données avec qui une donnée modifiée est en intersection. Cela nécessite de maintenir un graphe de ces intersections, et se répercute par un protocole de cohérence NP-complet. En revanche, garantir que chaque sous-donnée est indépendante est relativement aisé si l'on suit deux règles dans la conception des *filtres* :

1. Le *filtre* génère des sous-données disjointes.[4]
2. On ne manipule que les sous-données et pas les données intermédiaires.

En d'autres termes, cela revient à supposer que l'on peut représenter les données sous une forme arborescente, et que seule les feuilles de l'arbre sont accessibles. Appliquer un filtre consiste donc à créer un sous-arbre à la place du nœud qui modélise la donnée sur laquelle on applique ce filtre.

La première des deux règles impose des limites sur la liberté laissée au programmeur qui ajoute des *filtres*. En pratique, l'expérience apportée par HPF a démontré qu'un très grand nombre d'applications peuvent s'écrire avec des filtres qui sont des partitions. Cependant, pour un problème tel qu'une *convolution*, il est difficile de concevoir une distribution parfaitement disjointes. Il existe donc des situations où la représentation arborescente peut sembler limitée, mais il est toujours possible d'écrire un *filtre* où les intersections qui posent problème deviennent des sous-données à part entière : cette approche complique l'écriture de filtre, mais cela permet d'entretenir un protocole de cohérence très efficace plutôt que de se ramener à la résolution de problèmes NP-complet à chaque accès mémoire.

[4]Cela n'implique pas pour autant que ce soit une partition.

La seconde limitation nous semble également surmontable si l'on considère le cadre d'utilisation des *filtres* : conceptuellement, appliquer un *filtre* permet de raffiner la granularité sur laquelle opèrent les tâches. Ainsi il est assez naturel qu'à un instant donné, un algorithme n'accède qu'aux sous-données constituées par les feuilles. Il est cependant possible qu'un algorithme ait besoin d'accéder à des données qui ne sont pas des feuilles (typiquement dans le cadre d'une *réduction*). La solution la plus simple consiste à dés-appliquer les *filtres* jusqu'à ce que le nœud interne soit une feuille, puis à ré-appliquer ces *filtres* au besoin, mais cette approche nécessite de contrôler explicitement quels filtres sont appliqués. Ceci peut éventuellement être fait de manière transparente si la mise en œuvre du protocole de cohérence assure l'absence d'accès à un sous-arbre dont la racine est une donnée en cours d'utilisation.

Chapitre 4

Éléments d'implémentation

Dans ce chapitre, nous allons présenter quelques détails sur la mise en œuvre des propositions du chapitre précédent.

4.1 Organisation de la pile logicielle

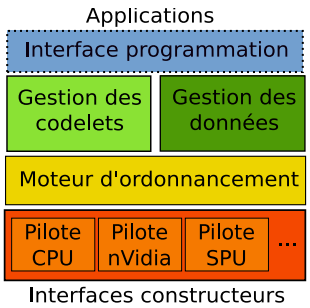


FIG. 4.1: Organisation de la pile logicielle

L'architecture du prototype de plateforme que nous avons mis en œuvre se veut modulaire comme le montre la Figure 4.1. Le bas de la pile est constitué des pilotes nécessaires aux différents types d'accélérateurs. Au cœur se trouve le moteur d'ordonnancement chargé de répartir les *codelets* sur les ressources de calculs à l'aide des mécanismes disponibles et qui est notamment appelé lorsqu'une application soumet une *codelet* au support exécutif. Au dessus, on dispose de deux bibliothèques, la première permet de soumettre des *codelets*, et la seconde permet de gérer les mouvements de données. On note enfin la présence d'une couche logicielle, pas encore implémentée, qui permet aux applications de masquer l'utilisation des *codelets* par une interface plus adaptée comme suggéré dans la Section 3.2.2.

La gestion des *codelets* est abordée dans la Section 4.2.1, la Section 4.2.2 présente l'interaction entre le cœur et les pilotes spécifiques, enfin nous présentons des éléments d'implémentation de la bibliothèque de gestion de données dans la Section 4.3.

4.2 Gestion des codelets

4.2.1 La structure de codelets

Nous avons jusque là présenté la *codelet* comme une structure de donnée modélisant une tâche de calcul. Cela se traduit *en substance* par :

```
1     typedef struct codelet_t {
2         enum {ANY, GPU, SPU, CORE} where;
3         cl_func core_func, cuda_func ... ;
4         buffer_descr buffers[];
5         callback cb;    /* do "cb(argcb)" when finished */
6     } codelet;
```

Cette structure indique plusieurs catégories d'informations :
- Il faut fournir les implémentations de la *codelet* pour les accélérateurs autorisés (`core_func` pour un cœur et `cuda_func` pour CUDA etc.). Nous utilisons un type abstrait `cl_func` pour représenter un code potentiellement déporté. [1]
- Le programmeur fournit une description de haut niveau des données manipulées par la *codelet* ainsi que le mode d'accès associé à chacune d'entre elles.
- La *codelet* se termine par l'exécution d'une continuation décrite par le type `callback`.

4.2.2 Distribuer les codelets

Dans ce prototype, nous adoptons un modèle très simple. Les applications peuvent soumettre des tâches (`push`) que les accélérateurs sont chargés d'exécuter. Lorsqu'un de ces derniers est inactif, il soumet une requête à l'ordonnanceur pour obtenir une tâche (`fetch`), et une fois celle-ci terminée, la continuation est exécutée (`callback`).

Mise en œuvre de pilotes spécifiques pour les accélérateurs

L'utilisation de pilotes *multithreadés* offre plusieurs avantages. D'une part, la soumission des tâches et l'exécution des continuations peuvent être découplés, ce qui est indispensable pour exploiter efficacement des accélérateurs *asynchrones*. D'autre part, il est possible de mieux utiliser les processeurs hôtes qui risquent d'être le goulet d'étranglement d'un modèle *maître-esclaves*. Concrètement, le pilote qui déporte du calcul sur les SPEs du CELL bénéficie de l'*hyperthreading* sur le processeur principal en dédiant un cœur (virtuel) au traitement des continuations alors que l'autre cœur est chargé de soumettre des tâches et d'exécuter le reste de l'application.

L'ordonnanceur

En pratique, notre ordonnanceur s'articule autour d'une file de tâches soumises par les applications d'un coté de la file alors que les pilotes piochent des tâches de l'autre coté. Lorsqu'une tâche ne peut être exécutée par un accélérateur qui l'a piochée, celle-ci est reposée au début de la file. Les travaux sur le langage CILK [7] démontrent que l'utilisation d'une file apporte certains avantages puisque les tâches sont exécutées selon un parcours en profondeur, ce qui permet d'exploiter les propriétés de localité. Le vol de tâche se fait selon un parcours

[1] Ce type peut se traduire par un simple pointeur de fonction pour un cœur, ou alors contenir des informations plus conséquentes, notamment des informations concernant le code à transférer sur l'accélérateur. La gestion du code en tant que donnée peut d'ailleurs se faire par le biais de notre bibliothèque.

en largeur, ce qui réduit les problèmes d'équilibrage de charge puisque l'on a tendance à voler des tâches de granularité importante.

En revanche, cette approche souffre d'un problème de *contention* dès lors que tous les pilotes entrent en compétition pour l'unique file, protégée par un verrou. Or si l'on considère que certains accélérateurs (comme les SPEs) ne peuvent manipuler cette structure que par le biais d'accès DMA, ce problème de verrouillage devient particulièrement gênant. Cette approche devra donc être modifiée par la suite, deux points semblent prometteurs dans le but de passer à l'échelle :

Construire une structure hiérarchique qui souffre naturellement de moins de contention. La plateforme BUBBLESCHED exploite d'ailleurs cette approche grâce à une représentation de la machine sous une forme arborescente [39]. Cependant, cela implique l'utilisation de mécanismes d'ordonnancement complexes tels que le vol de travail mis en œuvre par CILK [42].

Mettre en œuvre des algorithmes *lock-less* tels que l'algorithme THE utilisé par CILK 5 permet de protéger les accès à une file de tâches avec un faible surcoût en assurant que l'on peut détecter les accès concurrents.

4.3 Prototype de bibliothèque de gestion de données

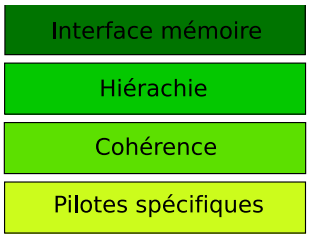


FIG. 4.2: Organisation de la bibliothèque de gestion de données

Dans cette section, nous présentons la structure générale de la bibliothèque de gestion des données qui est composée de plusieurs modules distincts comme le montre la Figure 4.2. Il n'y a pas d'*a priori* sur la représentation réelle des données dont on assure la cohérence, et la manipulation de données hiérarchiques se fait sur des structures d'arbres, et non pas sur les données elles-mêmes.

Un pilote spécifique est associé à chacun des types d'accélérateurs. La Section 4.3.1 montre plus en détails que cette couche ne nécessite que l'écriture de quelques primitives. La cohérence de chaque donnée est assurée par un automate qui met en œuvre le protocole MSI présenté dans la Section 4.3.2. L'utilisation des *filtres* dans la Section 4.3.3 permet de manipuler des données hiérarchiques. Comme suggéré par la Section 3.3.2, on peut faire coexister plusieurs interfaces mémoire pour adopter la représentation la plus adaptée.

4.3.1 Structure d'un pilote et notion de nœud mémoire

Chaque accélérateur est associé un *nœud mémoire*, c'est à dire une zone directement accessible par la ressource de calcul. En pratique, une carte graphique est associée à sa propre RAM, un SPE à sa mémoire locale, et un cœur à la mémoire principale.

Un pilote doit être capable de lire (ou d'écrire) des données d'un nœud depuis (ou vers) un autre nœud accessible de tous ; ou d'un nœud spécifique à un nœud autre que la mémoire principale lorsque c'est possible (eg. entre deux SPUs, ou entre deux GPUs dont les modèles

ou les constructeurs diffèrent). Au final, il faut fournir une primitive utilisée pour déplacer une donnée d'un nœud à un autre de manière transparente.

```
1 int driver_copy_data_1_to_1(data_state *data,
2                              uint32_t src_node, uint32_t dst_node);
```

Le pilote doit également fournir une primitive pour allouer de la mémoire. En effet, si on veut copier une donnée d'un nœud A à un nœud B, il peut être nécessaire d'allouer une zone mémoire sur B au préalable. Notre approche *paresseuse* permet une gestion de la mémoire qui se fait au besoin, ce qui évite de consommer les ressources tant que ce n'est pas strictement nécessaire. Il faut donc fournir une seconde primitive pour allouer de la mémoire sur un nœud :

```
1 int allocate_memory_on_node(data_state *data, uint32_t node);
```

4.3.2 Maintenir des données cohérentes

Implémentation du protocole de cache

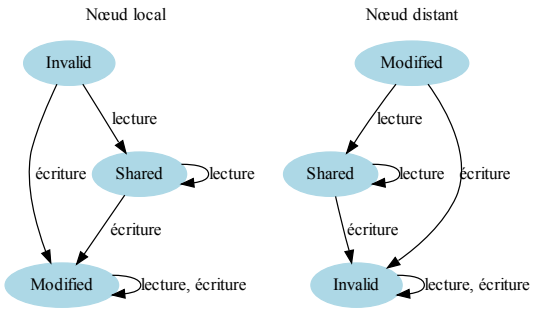


FIG. 4.3: Protocole MSI

Pour chaque donnée, et pour chaque nœud, on associe un état parmi les 3 possibles. La Figure 4.3 montre l'automate qui met cet état à jour en fonction du type d'accès. Il permet de mettre en œuvre un protocole de type MSI qui agit non seulement sur l'état de la donnée au niveau local, mais qui invalide aussi des copies distantes si nécessaire.

En plus du maintient de l'état de la donnée, ce protocole décrit les transferts nécessaires entre les différents nœuds mémoires pour aller chercher les données là où elles sont à jour. Il faut cependant remarquer que seule la lecture d'une donnée qui n'est pas disponible localement génère un tel transfert ; dans le reste des cas, seul l'état associé au nœud local est modifié.

Si une donnée est disponible sur plusieurs nœuds distants, il faut éventuellement faire un choix afin de minimiser le coût de transfert tout en évitant de toujours surcharger la même ressource.

Gestion de la concurrence des accès au cache

Plusieurs ressources de calcul peuvent faire appel à la bibliothèque de gestion de donnée simultanément. Il y a en effet deux facteurs de concurrence à prendre en compte :

1. Il faut protéger le descripteur de la donnée qui contient l'état associé à chaque nœud.
2. Il faut protéger le contenu de la donnée des accès concurrents au sein de l'application.

Nous associons le descripteur de chacune des données à un *mutex*. Les sections critiques qui consistent à mettre à jour l'état de chaque nœud étant *a priori* raisonnablement courtes, ce *mutex* utilise un mécanisme d'attente active.

Protéger la donnée elle-même nécessite une structure plus complexe : il est possible que plusieurs *codelets* accèdent à une même donnée sans pour autant la modifier. L'exclusion mutuelle n'est donc pas adaptée à ce contexte similaire au problème dit *des lecteurs et des écrivains*. Nous avons donc utilisé des verrous en lecture/écriture (ou *RW-locks*). Cette primitive permet de protéger les données de manière naturelle tout en autorisant des lectures concurrentes.

Gestion des défauts de capacité

L'utilisation de ces verrous en lecture/écriture plutôt qu'un simple verrouillage naïf permet également de maintenir un compteur de références sur chacune des données. Ainsi lorsqu'une *codelet* a besoin d'une donnée mais qu'il n'y a plus assez de mémoire disponible, nous avons mis en place un mécanisme d'éviction des données non utilisées, c'est à dire une stratégie de *ramasse-miettes* (ou *garbage-collector* en anglais).

Nous associons une liste de descripteurs à chaque nœud mémoire. Lors d'une allocation mémoire réussie, un descripteur est ajouté ; en revanche s'il n'y a plus de mémoire disponible, on parcourt cette liste afin de trouver des données qui ne sont plus utilisées[2]. Tout comme l'allocation, notre dés-allocation est donc paresseuse et nous maintenons une seconde liste pour chaque *nœud mémoire* pour stocker les descripteurs des données qui ne sont plus utilisés afin de les libérer au besoin.

4.3.3 Exprimer la hiérarchie

La notion de *filtre* introduite en Section 3.3.2 est une opération qui permet de partitionner une donnée en plusieurs sous-données. Ainsi si l'on suppose qu'une donnée peut se décrire complètement par quelques paramètres, il suffit de déterminer la transformation à faire subir à chacun de ces paramètres pour écrire un *filtre*.

La Figure 4.4 montre par exemple l'écriture d'un filtre qui effectue une distribution par blocs sur une interface de type BLAS. Il faut noter que certains paramètres n'ont pas de sens global mais doivent être calculés pour chaque *nœud mémoire* (typiquement l'adresse du premier élément).

Paramètre	Signification	Transformation à effectuer ($i^{\text{ième}}$ bloc)
nx	nombre d'éléments par ligne	$nx \leftarrow \frac{nx}{k}$
ny	nombre de lignes	$ny \leftarrow ny$
ptr	adresse du premier élement	$ptr \leftarrow ptr + i\frac{nx}{k}$
ld	distance entre deux lignes	$ld \leftarrow ld$

FIG. 4.4: Distribution par blocs

En appliquant des *filtres* sur une (sous-)donnée, on construit alors une structure arborescente dont on manipule les feuilles. Le calcul des paramètres *locaux* se fait là aussi de manière

[2]On ne dés-alloue jamais des données fournies par le programmeur dans un souci de transparence.

paresseuse. Il est d'ailleurs intéressant d'observer que la distance entre deux lignes est un paramètre local, ainsi il est possible de manipuler une zone de mémoire contigüe sur l'un des nœuds mémoires et pas sur les autres (ce qui est potentiellement bénéfique du point de vue de la localité et donc de l'utilisation du cache).

Chapitre 5

Évaluation

5.1 Environnement d'évaluation

Les expériences présentées dans cette section ont été principalement effectuées sur la machine BARRACUDA qui est un quadricœur XEON E5410 cadencé à 2.33GHZ. BARRACUDA dispose d'une carte graphique NVIDIA QUADRO FX 4600 compatible avec CUDA. Cette machine dispose par ailleurs de 4GO de RAM auxquels s'ajoutent 768MO de mémoire embarquée sur la carte graphique. Du point de vue logiciel, BARRACUDA s'appuie sur un noyau LINUX 2.6 et utilise CUDA 1.1. La mise en œuvre sur CELL se fait avec la machine LOOPING qui est une simple PLAYSTATION 3 utilisant également un noyau LINUX 2.6. Il n'est possible d'accéder qu'à 6 des 8 coprocesseurs et l'on ne dispose que de 256MO de RAM. La programmation sur cette plateforme se fait à l'aide du SDK 3.0 fourni par IBM.

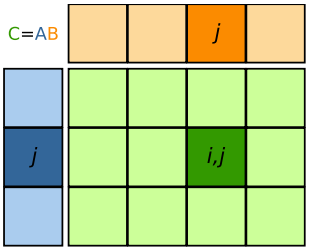
Enfin les machines HAGRID et BOB sont respectivement 8 dual-core AMD OPTERON 865 et un double quadricœur INTEL XEON E5345 ayant tous des processeurs cadencés à 2.33GHZ. Ces deux machines permettent de tester le comportement de notre code sur des machines multicœurs homogènes.

Nous utilisons la bibliothèque ATLAS 3.6.0, ainsi que CUBLAS qui est l'implémentation des routines BLAS fournie avec CUDA.

5.2 Cas du produit matriciel

Dans un premier temps, nous allons étudier le comportement de notre prototype sur l'exemple classique de la multiplication de matrice.

5.2.1 Programmabilité



Algorithme 5.1

1 **pour** i = 1 **à** $n_{colonnes}$ **faire**
2 **pour** j = 1 **à** n_{lignes} **faire**
3 $C_{ij} = A_j \times B_i$

FIG. 5.1: Multiplication de matrices par blocs

La Figure 5.2.1 décrit la parallélisation d'un produit matriciel. Nous allons montrer comment notre système permet de manipuler les données et comment il est possible de lancer les tâches qui correspondent au calcul de chacun des sous-blocs de la matrice résultante.

Partition des données

Dans notre cas, l'application doit partitionner la matrice A en `nslicesy` blocs, la matrice B en `nslicesx` blocs, et appliquer successivement ces deux opérations sur la matrice C. On suppose que le pointeur `pA` (resp. `pB` et `pC`) indique l'adresse du premier élement de A (resp. B et C), que cette matrice contient `nxA` éléments par lignes et `nyA` lignes qui sont distinctes de `ldA` éléments chacune.

```
1  data_state A_state, B_state, C_state;
2
3  /* déclaration des données A, B et C */
4  monitor_new_data(&A_state, pA, ldA, nxA, nyA, sizeof(float));
5  monitor_new_data(&B_state, pB, ldB, nxB, nyB, sizeof(float));
6  monitor_new_data(&C_state, pC, ldC, nxC, nyC, sizeof(float));
7
8  /* création d'un filtre horizontal */
9  filter f;
10   f.filter_func = block_filter_func;
11   f.filter_arg = nslicesx;
12
13 /* création d'un filtre vertical */
14 filter f2;
15   f2.filter_func = vertical_block_filter_func;
16   f2.filter_arg = nslicesy;
17
18 /* application des filtres sur A et B */
19 partition_data(&A_state, &f2);
20 partition_data(&B_state, &f);
21
22 /* application des deux filtres en cascade sur C */
23 map_filters(&C_state, 2, &f, &f2);
```

La primitive `monitor_new_data` va donc déclarer une donnée que la bibliothèque permet ensuite de manipuler à l'aide de la structure de donnée `data_state`. Il est alors possible de faire référence au bloc (i, j) de la matrice C à partir de cette structure de donnée grâce à la primitive `get_sub_data`.

L'utilisation des filtres est relativement aisée puisqu'il suffit de créer une structure `filter`, d'appeler une des fonctions de filtrage fournies par la bibliothèque, et de lui donner un éventuel argument comme ici le nombre de blocs. On applique alors ce filtre sur une donnée décrite par un `data_state` avec la primitive `partition_data`, ou `map_filters` qui applique un filtre sur toutes les feuilles de l'arborescence.

Lancer les codelets

Il est alors possible de traduire l'algorithme 5.2.1 de manière directe avec l'aide des *codelets* et de la description des données de haut niveau.

```
1  for (i = 0; i < nslicesx; i++) {
2    for (j = 0; j < nslicesy; j++) {
3      /* une tâche encapsule la codelet qui est réutilisable */
```

```
4       job_t jb = job_new();
5       codelet *cl = malloc(sizeof(codelet));
6       jb->cl = cl;
7
8       /* tout le monde peut exécuter la codelet */
9       jb->where = ANY;
10      cl->core_func = core_mult;
11      cl->cublas_func = cublas_mult;
12
13      /* que faire quand on a fini ? */
14      jb->cb = callback_func;
15      jb->argcb = &jobcounter;
16
17      /* il faut lire A[j] et B[i] puis écrire dans C[i,j] */
18      jb->nbuffers = 3;
19      jb->buffers[0].state = get_sub_data(&A_state, 1, j);
20      jb->buffers[0].mode = R;
21      jb->buffers[1].state = get_sub_data(&B_state, 1, i);
22      jb->buffers[1].mode = R;
23      jb->buffers[2].state = get_sub_data(&C_state, 2, i, j);
24      jb->buffers[2].mode = W;
25
26      push_task(jb);
27    }
28  }
```

Il n'y a aucune dépendance entre les *codelets* donc la continuation `callback_func` ne consiste qu'à décrémenter le compteur `jobcounter` jusqu'à une valeur nulle. L'écriture des noyaux de calcul se repose alors directement sur les implémentations de BLAS disponibles.

```
1  void cublas_mult(buffer_descr *d, void *arg) {
2    cublasSgemm('n', 'n', d[2].nx, d[2].ny, d[0].nx, 1.0, d[1].ptr,
3                  d[1].ld, d[0].ptr, d[0].ld, 0.0, d[2].ptr, d[2].ld);
4  }
5
6  void core_mult(buffer_descr *d, void *arg) {
7    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
8                 d[2].ny, d[2].nx, d[0].nx, 1.0, d[0].ptr, d[0].ld,
9                 d[1].ptr, d[1].ld, 0.0, d[2].ptr, d[2].ld);
10 }
```

Quand l'application passe une donnée en remplissant `jb->buffers[i]`, le noyau de calcul peut alors utiliser cette donnée par le biais de `d[i]` sans avoir à se préoccuper des mouvements de données sous-jacents.

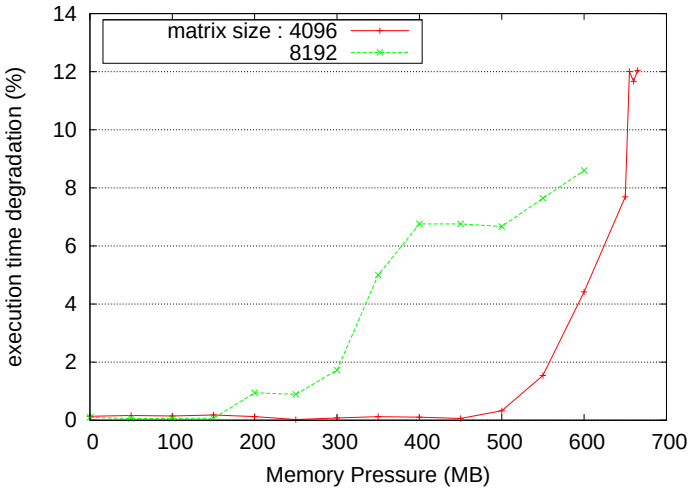
5.2.2 Pression mémoire

Dans l'hypothèse de l'utilisation d'une unique carte graphique pour faire du calcul, il est naturel de commencer par transférer le problème dans la mémoire de la carte, de faire le calcul, puis de ramener le résultat en mémoire principale. Cette approche souffre d'un certain nombre de limitations puisque le fait d'ajouter d'autres ressources de calcul impose de réécrire en partie l'algorithme, et il est également possible que le problème initial ne tienne tout simplement pas dans la mémoire de la carte.

Une solution naïve consiste à faire transiter les données traitées et celles produites entre chaque appel à un noyau de calcul afin de s'assurer que les données restent cohérentes de part

et d'autre de la machine. Cette approche, qui va à l'encontre de notre démarche *paresseuse*, ne peut *a priori* pas bénéficier d'une possible réutilisation des données entre les noyaux de calcul. La lenteur relative du bus mémoire par rapport au débit mémoire interne à la carte graphique, et le surcoût induit sur l'hôte et sur le bus rendent cette méthode assez rédhibitoire.

Dans notre approche, les mouvements de données se font au besoin, et les transferts effectués auraient donc été nécessaires. Elle permet l'utilisation de plusieurs ressources de calcul de manière transparente et rend l'exécution de calculs potentiellement plus grands que la mémoire de la carte graphique possible.



taille des matrices	8192	16384
taille du problème	768 Mo	3072 Mo
référence	16,4s	138s
- 350 Mo	17,2s	140s

FIG. 5.2: Contourner les limitations mémoire

FIG. 5.3: Impact de la pression mémoire pour deux tailles de matrices

Afin d'étudier le comportement de notre approche vis-à-vis de la pression mémoire, nous avons pré-alloué une quantité déterminée de mémoire sur la carte graphique. Nous simulons ainsi le comportement d'une carte disposant de moins de mémoire, ou l'utilisation d'une carte par plusieurs applications se partageant la mémoire. La Figure 5.3 montre qu'il est possible d'effectuer le produit de matrices dont la taille excède substantiellement celle de la mémoire disponible sur la carte graphique (768 Mo).

Puisque nous sommes capables d'effectuer des calculs plus larges que la mémoire disponible, nous avons répété les mêmes expériences en limitant la mémoire de moitié comme sur la Figure 5.3 où l'impact sur le temps d'exécution est ici de l'ordre de 5% sachant que chaque *codelet* manipule 3 blocs de 4Mo chacun, et que seule une unique carte graphique est utilisée pour réaliser ce calcul.

La Figure 5.2.2 montre l'absence de dégradation des performances tant que le problème est contenu dans la mémoire disponible. et que seule une très forte pression mémoire peut causer une détérioration substantielle, ici de l'ordre de 10% quand la mémoire n'excède plus que de peu la taille minimale de 3 blocs nécessaires à chaque *codelet*.

Ce résultat est important puisque les différents accélérateurs disposent de mémoires embarquées dont la taille varie de plusieurs ordres de grandeur. Il faut donc que les limitations mémoires soient aussi transparentes que possible. Or nous voyons ici que l'on peut lancer le calcul sans le modifier pour tenir compte de la mémoire disponible sur telle ou telle carte graphique.

5.2.3 Hétérogénéité

Nous avons comparé les performances de notre multiplication de matrices pour différentes configurations afin de mettre en évidence l'intérêt de faire collaborer plusieurs ressources de calcul.

Plutôt que de comparer des accélérations par rapport à un seul processeur qui n'ont finalement que peu de signification dans un contexte hétérogène, nous avons utilisé la densité de calcul ainsi que le débit mémoire comme métriques. Nous évitons ainsi l'eccueil consistant à comparer une mauvaise implémentation sur une architecture face à un code optimisé sur une autre, décuplant le gain supposé de manière totalement artificielle. Le manque de support efficace pour extraire ces informations des compteurs matériels nous pousse toutefois à calculer ces paramètres de manière synthétique pour des problèmes simples.

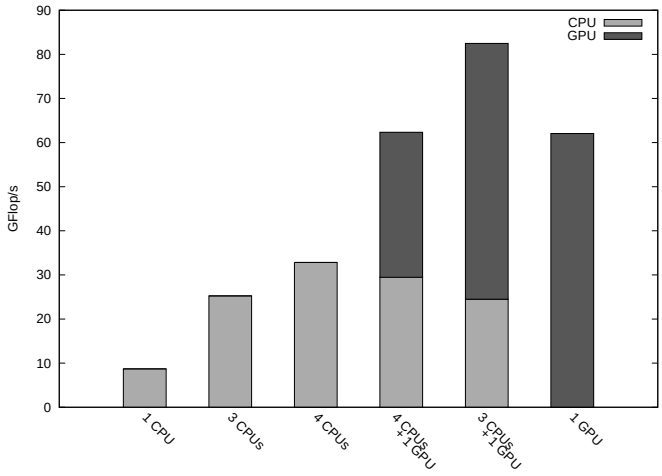


FIG. 5.4: Combiner des ressources de calcul hétérogènes

Le résultats de la Figure 5.4 nous fournissent plusieurs informations : d'une part, il y a effectivement un avantage à utiliser le processeur en appoint de la carte graphique, et d'autre part on observe une forte dégradation des performances si l'on utilise la carte graphique et tous les cœurs. En effet, un cœur est alors chargé de contrôler la carte graphique et d'effectuer un calcul particulièrement sensible aux effets de cache. Il faut donc être particulièrement vigilant dans le placement des calculs, et le sacrifice d'une ressource de calcul peut s'avérer utile. On remarque par ailleurs que dans la configuration optimale, on obtient 82.47GFLOPS avec trois cœurs et une carte graphique, c'est à dire 95% de la somme des performances obtenues individuellement par trois cœurs (25.24GFLOPS) et une carte graphique(62.06GFLOPS).

5.2.4 Équilibrage de charge

Pour améliorer les performances, il nous fallait d'abord les comprendre. Pour ce faire, nous avons utilisé la bibliothèque FXT [43] qui permet de tracer le comportement d'une application de manière peu invasive, et nous utilisons ces traces pour générer un diagramme de GANTT de l'exécution des *codelets*.

La Figure 5.5 montre le comportement d'un produit matriciel effectué simultanément sur 3 cœurs et sur une carte graphique, sachant que les performances varient fortement selon la taille des blocs utilisés. Chacune des 16 tâches implique la même quantité de calcul, mais la carte graphique peut les effectuer bien plus rapidement que les cœurs du processeur. Nous avons

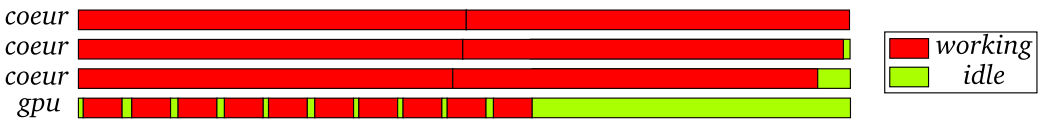


FIG. 5.5: Problème d'équilibrage de charge

ici une preuve manifeste d'un mauvais équilibrage de charge lié à l'ordonnanceur *glouton*, ce qui confirme la nécessité d'utiliser des politiques d'ordonnancement plus évoluées que nous avons présenté dans la Section 3.1.3. On peut toutefois noter que le choix d'une granularité plus faible permettrait de réduire l'impact des problèmes d'équilibrage de charge dans ce cas précis.

5.3 Résolution de l'équation de la chaleur

Afin d'utiliser notre prototype sur un exemple un peu plus concret qu'une simple multiplication matricielle, nous avons mis en œuvre une résolution numérique de l'équation de la chaleur.

5.3.1 Méthodes des éléments finis et formulation matricielle

L'Annexe B montre qu'il est possible de formuler ce problème sous la forme d'un système linéaire à résoudre. Pour ce faire, il faut tout d'abord créer un maillage du domaine sur lequel on résout le problème. On en déduit un ensemble de fonctions *tests* qui forme la base d'un espace d'interpolation dans lequel on va chercher à résoudre le problème initial.

La première phase de cette application, appelée *assemblage*, consiste à calculer les matrices qui décrivent le système à résoudre. Cette étape relativement peu coûteuse est constituée d'une succession d'intégrations numériques.

La résolution du système linéaire forme alors la seconde phase de l'application, qui est généralement bien plus coûteuse que l'assemblage. Nous avons choisi d'utiliser une factorisation LU qui permet de ré-écrire la *matrice de rigidité* K sous la forme d'un produit de matrices triangulaires L et U respectivement inférieure et supérieure.

Notre but n'est cependant pas d'écrire la factorisation LU la plus efficace possible, mais de concevoir une application qui met en évidence les besoins et les limitations de notre prototype. Par conséquent nous ne prenons pas en compte les caractéristiques de la matrice de rigidité, notamment nous ne considérons pas son caractère creux et symétrique. Nous négligeons également les problèmes de stabilité numérique. Nous allons donc concevoir une factorisation de matrices denses à l'aide de notre prototype de support exécutif.

5.3.2 Factorisation LU par blocs

La Figure 5.3.2 montre que la parallélisation de la factorisation LU peut se faire à l'aide d'un découpage par blocs. Cet algorithme consiste en un *pipeline* de 4 types de *codelets*, que l'on désignera par 11, 12, 21 et 22 par la suite.

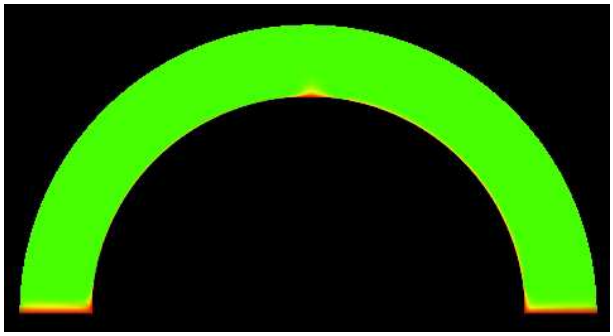


FIG. 5.6: Flux de chaleur ascendant sur une arche

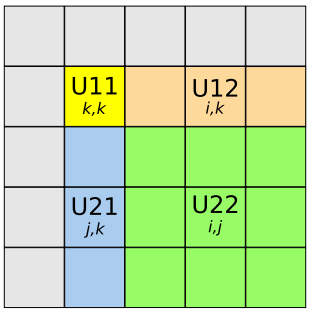


FIG. 5.7: Factorisation LU par blocs

Algorithme 5.2

pour k = 1 **à** n **faire**
  Factoriser $U11_{(k,k)}$
  pour i = (k+1) **à** n **faire**
    résoudre $U11_{(k,k)}X = U12_{(i,k)}$
    $U12_{(i,k)} \leftarrow X$
  pour j = (k+1) **à** n **faire**
    résoudre $XU11_{(k,k)} = U21_{(k,j)}$
    $U21_{(k,j)} \leftarrow X$
  pour i = (k+1) **à** n **faire**
    pour j = (k+1) **à** n **faire**
      $U22_{(i,j)} -= U12_{(i,k)}U21_{(k,j)}$

5.3.3 Extraire suffisamment de parallélisme

La Figure 5.8 met en évidence un potentiel manque de parallélisme. En effet, à l'itération k, si les $2\,(n-k-1)$ tâches 12 et 21 ainsi que les $(n-k-1)^2$ tâches 22 peuvent se faire en parallèle, la tâche 11 est exécutée de manière séquentielle, ce qui a un impact direct sur l'accélération selon la loi de AMDAHL.Nous avons donc ré-exprimé les dépendances de manière plus fine comme sur la Figure 5.9 afin d'extraire autant de parallélisme que l'algorithme le permet. Si on note $A \leftarrow B$ le fait que A dépend de B, et C^k la *codelet* de type C à l'itération k.

Sur la Figure 5.10, on observe les diagrammes de GANTT obtenus pour chacune des deux versions : le premier graphe représente le travail de 3 cœurs et d'une carte graphique, on voit très nettement que le travail des cœurs est entrecoupé de longues périodes d'inactivité ; le graphe qui suit est obtenu avec une meilleure gestion des dépendances entre tâches, et on voit bien que toutes les ressources de calcul sont occupées en permanence jusqu'à la fin du programme où l'algorithme souffre d'un manque inhérent de parallélisme. La durée de calcul est ici réduite de 15% dès lors que l'on applique cette optimisation. Ce résultat confirme donc la nécessité de fournir un moyen de structurer l'application en exprimant les dépendances entre tâches plutôt que de laisser le programmeur s'en charger par ses propres moyens, généralement dans les continuations.

5.3.4 Besoin d'un ordonnancemnt avec des priorités

En relâchant les relations de dépendances, nous avons observé un gain substantiel grâce à l'utilisation quasi ininterrompue des ressources de calcul. Cependant, nous avons remarqué qu'il est tout à fait possible de manquer de parallélisme si les tâches de type 11 ne sont pas effectuées en priorité puisque toutes les tâches de l'itération k ne peuvent se lancer tant que

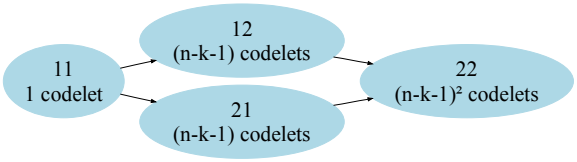


FIG. 5.8: Pipeline naïf

1. $U11^k_{k,k} \leftarrow U22^{k-1}_{k,k}$
2. $U12^k_{i,k} \leftarrow U22^{k-1}_{i,k} + U11^k_{k,k}$
3. $U21^k_{k,j} \leftarrow U22^{k-1}_{k,k} + U11^k_{k,k}$
4. $U22^k_{i,j} \leftarrow U22^{k-1}_{i,j} + U12^k_{i,k} + U21^k_{k,j}$

FIG. 5.9: Dépendances réelles

la *codelet* $U11^k_{k,k}$ n'est pas terminée. Nous avons donc mis en œuvre une notion de priorité dans les relations de dépendances : il est souhaitable que les tâches $U11^k_{k,k}$ (ainsi que $U22^{k-1}_{k,k}$, $U12^{k-1}_{k,k-1}$ et $U21^{k-1}_{k-1,k}$ dont $U11^k_{k,k}$ dépend directement) soient effectuées le plus tôt possible.

Lorsque l'application soumet une tâche par un appel à `push_task`, celle-ci est placée à la fin de la file de *codelets* à exécuter, alors que les ressources de calcul piochent des *codelets* en tête de cette même file. Par conséquent, nous avons simplement créé une primitive `push_prio_task` qui place la tâche en tête de la file. Cette approche est limitée puisqu'elle n'offre pas de garantie d'un respect strict des priorités, et qu'il n'est pas possible de mettre en place des niveaux de priorités.

La Figure 5.11 met ainsi en évidence les gains obtenus en appliquant diverses optimisations sur notre programme de factorisation LU utilisant une carte graphique et un quadricœur ayant un cœur dédié. La première de ces optimisations consiste à punaiser la mémoire afin d'utiliser efficacement le moteur DMA de la carte graphique, et permet un gain généralement entre 5 et 10% sur le temps de calcul. La seconde optimisation consiste à casser le pipeline afin d'extraire suffisamment de parallélisme, on observe alors un gain substantiel de 20 à 30% par rapport à l'optimisation précédente. Enfin la dernière optimisation consiste à utiliser des priorités et permet d'améliorer le temps d'exécution de l'ordre de 5% à nouveau. Cette expérience nous démontre donc qu'il est indispensable de fournir des outils pour permettre au programmeur d'exprimer la structure de son algorithme, et qu'un bon ordonnancement permet ici de gagner entre 25 et 30% sur le temps de calcul.

Disposant donc d'un support exécutif qui nous permet d'appliquer un certain nombre d'optimisation algorithmiques de manière relativement simple, nous avons alors mesuré les performances obtenues sur un quadricœur assisté d'une carte graphique. La Figure 5.12 montre l'intérêt de choisir une granularité adaptée, et confirme que déterminer automatiquement la meilleure granularité reste un problème très difficile, notamment dans le cas d'une machine hétérogène comme c'est ici le cas.

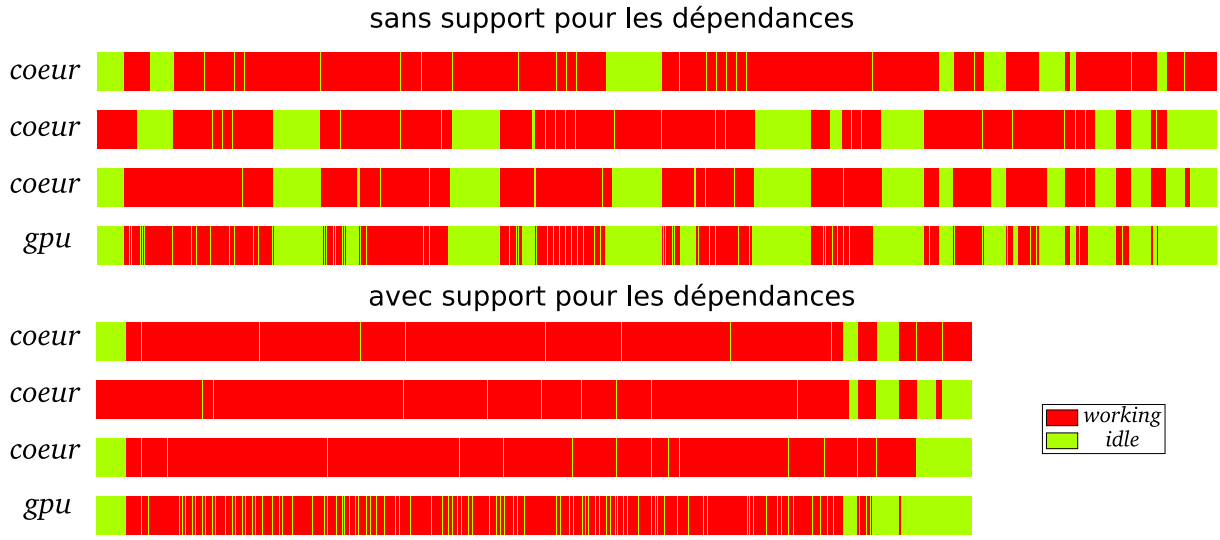


FIG. 5.10: Casser le pipeline pour gagner du parallélisme

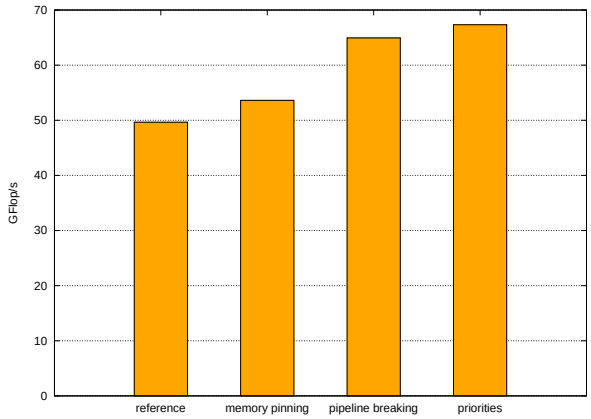


FIG. 5.11: Optimisation de la factorisation LU

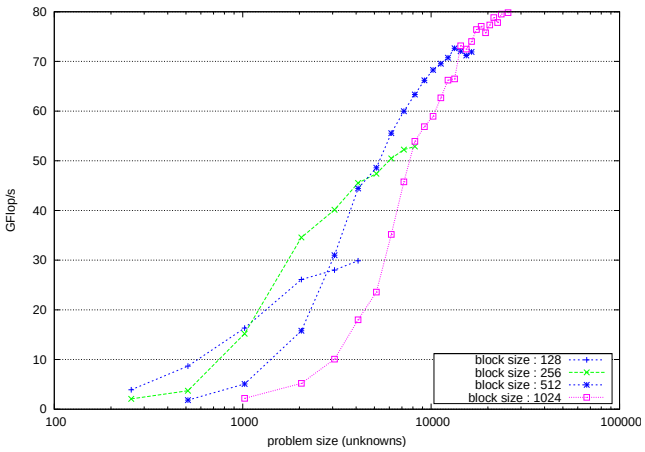


FIG. 5.12: Impact de la granularité

Chapitre 6

Conclusion

Dans ce document, nous avons tout d'abord conçu une bibliothèque pour manipuler des données distribuées sur une machine hétérogène. Grâce à une interface très expressive, nous pouvons donc utiliser des données hiérarchiques de manière transparente, laissant à la bibliothèque le soin de gérer les problèmes de concurrence. Nous avons ensuite proposé d'organiser nos calculs avec une structure appelée *codelet* que nous utilisons pour définir notre modèle d'exécution. Une tâche est décrite d'une part par une description des données manipulées, ainsi que leur mode d'accès, puis par les méthodes à appeler sur les différents accélérateurs, et enfin une éventuelle continuation. Les dépendances entre les *codelets* permettent au programmeur d'expliciter la structure même de l'algorithme.

Nous avons conçu un prototype pour des plateformes multicœurs et des cartes graphiques NVIDIA, et le portage sur CELL est en cours. Les premiers résultats sont très prometteurs puisque nous sommes en mesure d'écrire des codes très simples qui exploitent simultanném ent les cœurs d'un processeur et une éventuelle carte graphique. Disposant d'une représentation très structurée des calculs et des données, notre approche permet au programmeur de traduire un algorithme de manière directe sans avoir à se soucier des détails purement techniques. Nous avons ainsi démontré qu'il est possible d'écrire une factorisation LU qui exploite une plateforme hétérogène de manière transparente, et qu'il est possible de se concentrer sur des aspects purement algorithmiques, laissant notre support exécutif se charger des interactions entre les éléments de la machine, masquant notamment tous les transferts de données qu'il aurait été bien peu productif de gérer explicitement. Notre approche paresseuse fait que nous ne sommes plus limités par la taille de la mémoire des accélérateurs, ainsi il est possible de résoudre des problèmes dont la taille dépasse largement celle des mémoires des cartes graphiques par exemple.

Disposant donc d'applications simples, nous avons alors montré l'importance d'un ordonnancement de qualité pour exploiter les machines hétérogènes, notamment en fournissant suffisamment de parallélisme pour occuper les multiples cœurs vectoriels. Structurer l'application permet aussi au programmeur d'exprimer les informations qu'il connaît sur l'algorithme lui-même : exprimer les dépendances entre *codelets* permet ainsi de fournir une vision plus globale que la simple exécution d'une pile de tâches, ce qui ouvre la possibilité d'un ordonnancement bien plus évolué. À l'aide de mécanismes d'ordonnancement simples, il est possible de concevoir des stratégies d'ordonnancement évoluées, adaptées aux besoin spécifiques de l'application.

Ces prototypes devraient permettre d'exploiter des machines dotées de plusieurs accélérateurs (eg. plusieurs cartes graphiques) de manière transparente pour l'application. Le portage sur CELL pose encore des problèmes intéressants puisqu'il nous faut nous assurer que toutes nos structures de données sont manipulables avec des accès DMA. Cela nous amène natu-

rellement à la possibilité de fournir une bibliothèque de gestion des données avec une interface asynchrone, autorisant par exemple des mécanismes de *prefetch*. Un tel asynchronisme est d'ailleurs crucial pour exploiter pleinement les architectures sur lesquelles les accès mémoires peuvent être recouvert par du calcul, masquant ainsi le coût des transferts de données qui peut être critique pour les performances d'une application. Une meilleure modélisation de la machine permettra de prendre en compte des effets de type NUMA, laissant ainsi la possibilité de réduire autant que possible le coût d'exécution d'une *codelet* en la plaçant au mieux sur la machine du point de vue des données manipulées.

La structure de *codelet* pourra être utilisée pour maintenir des informations sur le comportement réel de l'application : cela permettra d'ordonnancer une *codelet* là où elle s'est révélée le plus efficace par le passé, et un modèle de coût permet de distribuer les *codelets* selon la durée de leur exécution.

Disposant des indications de haut niveau fournies par le programmeur, il sera alors possible de mettre en œuvre un panel de stratégies d'ordonnancement évoluées, que le programmeur sélectionnera selon les caractéristiques de son application. Face à l'utilisation d'accélérateurs et de processeurs multicœurs hétérogènes au sein des *grappes* et des super-calculateurs, il sera indispensable de chercher à adapter ces travaux sur des machines inter-connectées. Ainsi il faudra s'intéresser à des modèles de programmation hybrides, à l'image de ce que l'on observe actuellement autour de l'interaction entre MPI et OPENMP.

Finalement, nous espérons que notre support exécutif sera à même d'aider les environnement de compilation, offrant aux langages la possibilité de se consacrer à la génération de code. Nous sommes donc particulièrement attentifs à la possibilité d'un enrichissement des standards tels qu'OPENMP pour permettre de générer de *codelets*.

Annexe A

Exemples d'architectures multicœurs

A.1 Le processeur CELL

Conçu par IBM, SONY et TOSHIBA entre 2001 et 2006, le processeur CELL propose une approche en rupture avec les autres architectures multicœurs. Le CELL est une puce hétérogène principalement constituée du PPE et de 8 coprocesseurs appelées SPE. Le PPE, qui est le cœur principal, est une puce *hyperthreadée* basée sur une architecture de type POWER. Les SPE (*Synergistic Processing Element*) sont des cœurs de type RISC dont le but est d'accélérer les calculs du PPE, notamment grâce aux instructions vectorielles. Chaque SPE dispose d'une mémoire locale (*Local Store*) de 256KO, qui contient l'ensemble des données et du code accessible par le SPE. Les mouvements de données entre les cœurs se font par l'intermédiaire du MFC disponible sur chaque cœur (*Memory Flow Controler*), qui contient principalement un contrôleur DMA. Les transferts DMA sont alors véhiculés par l'EIB (*Element Interconnect Bus*) qui est un bus constitué de 4 anneaux concentriques.

Chaque SPE est donc un véritable processeur autonome, même si le système d'exploitation ne tourne que sur le PPE, et il est important de noter que si tous les cœurs peuvent échanger des données, ces mouvements se font par des mécanismes logiciels explicites qui peuvent rendre la programmation du CELL particulièrement technique. Tout comme le PPE peut être programmé à l'aide de l'interface PTHREAD, chaque SPE est contrôlé par la bibliothèque LIBSPE qui permet le contrôle d'un contexte de calcul par coprocesseur.

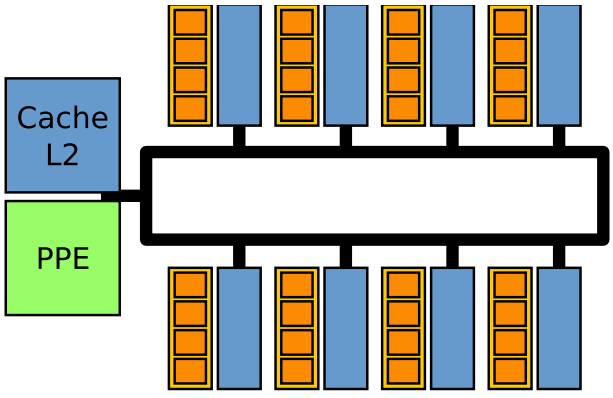


FIG. A.1: Schema de principe du processeur CELL

A.2 Processeurs graphiques

Contrairement au CELL, la conception des processeurs graphiques est naturellement influencée par des besoins spécifiques. Cela se traduit par des architectures vectorielles qui accèdent à la mémoire selon des motifs bien particuliers.

L'unité de contrôle réparti les instructions sur les différentes processeurs vectoriels, masquant les latences mémoires avec des changements de contextes particulièrement rapides. Le programmeur n'a cependant aucun contrôle sur ces changements de contexte dont l'efficacité repose justement sur une implémentation purement matérielle. Il faut donc fournir un parallélisme massif (eg. plusieurs dizaines de milliers de *threads*) pour exploiter ces architectures avec efficacité.

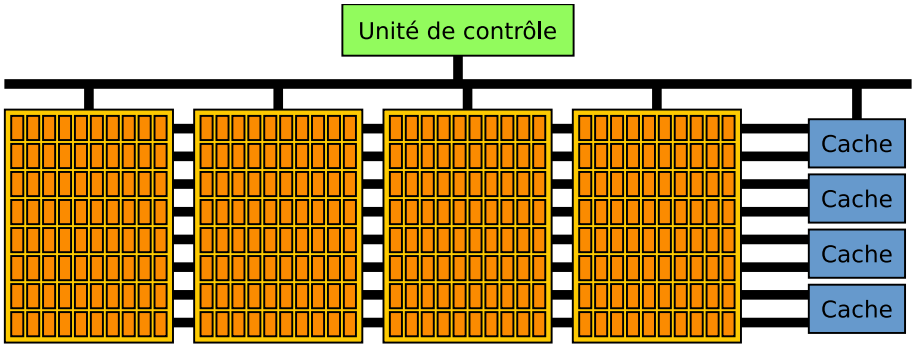


FIG. A.2: schéma de principe d'une ATI R600

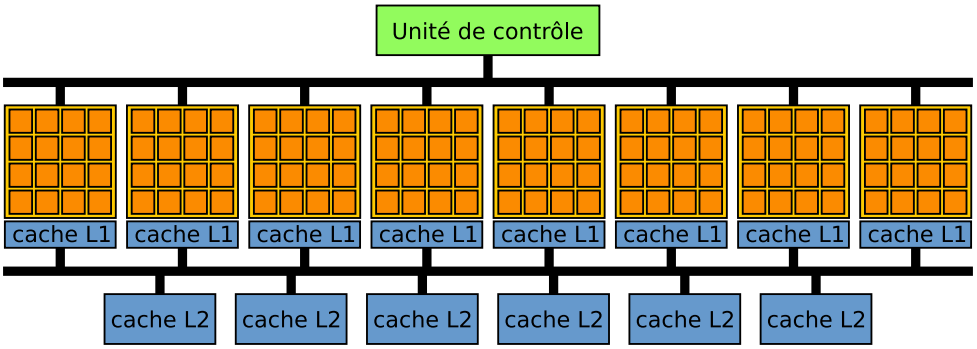


FIG. A.3: Schéma de principe d'une NVIDIA GEFORCE 8800

Architecture	Coprocesseurs	SIMD	Fréquence	Mémoire locale	Gestion des contextes
CELL	8	4	3.2GHZ	256KO	Manuelle
NVIDIA 8800	de 8 à 16	16×2	750×2MHZ	16KO + 8192 reg.	Matérielle

Annexe B

Résolution de l'équation de la chaleur par la méthode des éléments finis

B.1 Formulation du problème

Nous cherchons ici à connaître la température T, en tout point, d'un domaine connexe $\mathcal{D}$ dont la température est connue en ses bornes $\mathcal{C}$. On se place ici dans un domaine 2D mais les résultats sont généralisables à d'autres espaces.

L'équation de la chaleur en mode stationnaire est une équation de LAPLACE si l'on n'a pas de terme de flux. Ainsi, il faut vérifier que

$$\begin{cases} \nabla^2 T = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \\ T_{|\mathcal{C}} \text{ fixé} \end{cases}$$

B.2 Espace d'interpolation

La méthode des éléments finis consiste à résoudre le problème dans un espace de dimension finie afin de raisonner sur des interpolations [44]. Puisque l'espace des fonctions $u : \mathcal{D} \to \mathbb{R}$ n'est pas fini, la première étape consiste donc à construire un sous-espace de dimension fini. Les critères de choix d'un espace *convenable* sont totalement hors du cadre de ce document et on pourra se référer au cours d'ALLAIRE [45] pour de plus amples détails sur ce problème, de même on ne cherche pas à démontrer dans quelle mesure la fonction ainsi approximée sera *proche* de l'unique solution dont on admet également la solution, et on admet que toutes les conditions de continuité et de dérivabilité nécessaires aux calculs qui vont suivre sont vérifiées.

Une approche simple à mettre en œuvre une interpolation de LAGRANGE sur le domaine dont on suppose disposer d'un maillage conforme $\mathcal{M}$ dont les n sommets sont notés m_i. On peut alors construire une base de cet espace à partir des fonctions $\{\psi_i\}_{i<n}$ telles que :

$$\forall i, j < n, \begin{cases} \psi_i(m_j) = \delta_i^j \\ \psi_i \text{ polynôme de degré 1 par morceau sur } \mathcal{D} \end{cases}$$

Toute fonction de l'espace interpolé est donc une combinaison linéaire de cette base et la projection de $f : \mathcal{D} \to \mathbb{R}$ dans cet espace est alors triviale puisque l'interpolation $\tilde{f}$ est simplement :

$$\tilde{f} = \sum_{i=0}^{n-1} f(m_i)\,\psi_i$$

B.3 Forme faible

Le problème de LAPLACE étant un exemple classique de la mise sous forme variationnelle, il apparaît dans de nombreuses introductions aux éléments finis [46]. Nous allons ré-exprimer le problème afin de chercher une solution u dans l'espace d'interpolation, ce qui implique par exemple de ne garder que des termes d'ordre au plus 1 puisque les interpolations sont linéaires par morceaux. Si $\nabla^2 u = 0$ alors,

$$\forall \phi : \mathcal{D} \to \mathbb{R}, \quad \int_{\mathcal{D}} \psi \nabla^2 u = 0$$

Étant donné que,

$$\int_{\mathcal{D}} \phi \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = \int_{\mathcal{D}} \frac{\partial}{\partial x} \left(\phi \frac{\partial u}{\partial x} \right) + \frac{\partial}{\partial y} \left(\phi \frac{\partial u}{\partial y} \right) - \frac{\partial \phi}{\partial x} \frac{\partial u}{\partial x} - \frac{\partial \phi}{\partial y} \frac{\partial u}{\partial y}$$

Et,

$$\frac{\partial}{\partial x_i} \left(\phi \frac{\partial u}{\partial x_i} \right) = \frac{\partial \phi}{\partial x_i} \frac{\partial u}{\partial x_i} + \phi \frac{\partial^2 u}{\partial {x_i}^2} \text{ pour } x_i \in \{x, y\}$$

Et que d'après théorème de divergence de GAUSS qui permet d'obtenir, si $\overrightarrow{n}$ est la normale au contour $\mathcal{C}$ et $\left(\overrightarrow{i}, \overrightarrow{j} \right)$ une base de $\mathcal{D}$:

$$\int_{\mathcal{D}} \frac{\partial}{\partial x} \left(\phi \frac{\partial u}{\partial x} \right) = \oint_{\mathcal{C}} \phi \frac{\partial u}{\partial x} \overrightarrow{i} \cdot \overrightarrow{n} \text{ et } \int_{\mathcal{D}} \frac{\partial}{\partial y} \left(\phi \frac{\partial u}{\partial y} \right) = \oint_{\mathcal{C}} \phi \frac{\partial u}{\partial y} \overrightarrow{j} \cdot \overrightarrow{n}$$

On obtient la forme faible du problème après avoir défini le terme de flux $q : \mathcal{C} \to \mathbb{R}$ tel que $q = \left(\frac{\partial u}{\partial x} \overrightarrow{i} + \frac{\partial u}{\partial y} \overrightarrow{j} \right) \cdot \overrightarrow{n}$:

$$\forall \phi : \mathcal{D} \to \mathbb{R}, \quad \int_{\mathcal{D}} \frac{\partial \phi}{\partial x} \frac{\partial u}{\partial x} + \frac{\partial \phi}{\partial y} \frac{\partial u}{\partial y} = \oint_{\mathcal{C}} \phi q$$

Si on cherche une solution dans l'espace d'interpolation avec $u = \sum_{i=0}^{n} u_i \psi_i$, on peut alors ré-exprimer le problème.

$$\forall \phi : \mathcal{D} \to \mathbb{R}, \quad \sum_{j=0}^{n-1} u_j \int_{\mathcal{D}} \left(\frac{\partial \phi}{\partial x} \frac{\partial \psi_j}{\partial x} + \frac{\partial \phi}{\partial y} \frac{\partial \psi_j}{\partial y} \right) = \oint_{\mathcal{C}} \phi q$$

Les éléments de la base de l'espace d'interpolation sont également appelées fonction *tests*, puisque l'on a en particulier :

$$\forall i < n, \quad \sum_{j=0}^{n-1} u_j \int_{\mathcal{D}} \left(\frac{\partial \psi_i}{\partial x} \frac{\partial \psi_j}{\partial x} + \frac{\partial \psi_i}{\partial y} \frac{\partial \psi_j}{\partial y} \right) = \oint_{\mathcal{C}} \psi_i q$$

On a donc un système d'équations que l'on peut alors ré-écrire sous forme matricielle. On défini deux matrices, une première matrice K généralement appelée matrice de *rigidité* (*stiffness matrix* en anglais), ainsi qu'une matrice Q qui contient les termes de flux. Si on pose $U = (\, u_0, \ldots, u_{n-1})$, il faut donc finalement résoudre :

$$K_{ij} = \int_{\mathcal{D}} \left(\frac{\partial \psi_i}{\partial x} \frac{\partial \psi_j}{\partial x} + \frac{\partial \psi_i}{\partial y} \frac{\partial \psi_j}{\partial y} \right)$$

$$Q_j = \oint_{\mathcal{C}} \psi_j \left(\frac{\partial \psi_j}{\partial x} \overrightarrow{i} + \frac{\partial \psi_j}{\partial y} \overrightarrow{j} \right) \cdot \overrightarrow{n}$$

$$KU = Q$$

Bibliographie

[1] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall : Implications of the obvious. *Computer Architecture News*, 23(1) :20–24, 1995.

[2] David Geer. Industry trends : Chip makers turn to multicore processors. *Computer*, 38(5) :11–13, 2005.

[3] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5) :589–604, 2005.

[4] Shekhar Borkar, Norman P. Jouppi, and Per Stenstrom. Microprocessors in the era of terascale integration. In *DATE '07 : Proceedings of the conference on Design, automation and test in Europe*, pages 237–242, San Jose, CA, USA, 2007. EDA Consortium.

[5] Franck Cappello and Daniel Etiemble. Mpi versus mpi+openmp on ibm sp for the nas benchmarks. In *Supercomputing '00 : Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 12, Washington, DC, USA, 2000. IEEE Computer Society.

[6] Intel threading building blocks 2.0 for open source.

[7] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *PLDI '98 : Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM.

[8] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1) :80–113, 03 2007.

[9] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg : a system for programming graphics hardware in a c-like language. In *SIGGRAPH '03 : ACM SIGGRAPH 2003 Papers*, pages 896–907, New York, NY, USA, 2003. ACM.

[10] Randi J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2005.

[11] Avi Bleiweiss and Arcot Preetham. Ashli - advanced shading langage interface, 2003. http ://ati.amd.com/developer/techreports/2003/Bleiweiss-AshliNotes.pdf.

[12] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. In *SIGGRAPH '04 : ACM SIGGRAPH 2004 Papers*, pages 787–795, New York, NY, USA, 2004. ACM.

[13] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus : stream computing on graphics hardware. In *SIGGRAPH '04 : ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

[14] Patrick McCormick, Jeff Inman, James Ahrens, Jamaludin Mohd-Yusof, Greg Roth, and Sharen Cummins. Scout : a data-parallel programming language for graphics processors. *Parallel Comput.*, 33(10-11) :648–662, 2007.

[15] Aaron E. Lefohn, Shubhabrata Sengupta, Joe Kniss, Robert Strzodka, and John D. Owens. Glift : Generic, efficient, random-access gpu data structures. *ACM Trans. Graph.*, 25(1) :60–99, 2006.

[16] CTM guide - CTI technical reference manual.

[17] Cuda zone. http ://www.nvidia.com/cuda.

[18] AMD FireStream SDK whitepaper. http ://ati.amd.com/technology/streamcomputing/.

[19] Gpu-tech. http ://www.gputech.com.

[20] *Programming the Cell Broadband Engine Examples and Best Practices*. http ://www.redbooks.ibm.com/abstracts/sg247575.html.

[21] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice boltzmann simulation optimization on leading multicore platforms. In *International Parallel And Distributed Processing Symposium (IPDPS)*, Miami, 2008.

[22] Catherine H. Crawford, Paul Henning, Michael Kistler, and Cornell Wright. Accelerating computing with the cell broadband engine processor. In *CF '08 : Proceedings of the 2008 conference on Computing frontiers*, pages 3–12, New York, NY, USA, 2008. ACM.

[23] David Kunzman, Gengbin Zheng, Eric Bohm, and Laxmikant V. Kalé. Charm++, Offload API, and the Cell Processor. In *Proceedings of the Workshop on Programming Models for Ubiquitous Parallelism*, Seattle, WA, USA, September 2006.

[24] B. Bouzas, R. Cooper, J. Greene, M. Pepe, and M. J. Prelle. Multicore framework : An api for programming heterogeneous multicore processors. In *Proc. of First Workshop on Software Tools for Multi-Core Systems*, New York, NY, USA, 2006. Mercury Computer Systems.

[25] Manabu Morita, Takahiro Machino, Minyi Guo, and Guojun Wang. Design and implementation of stream processing system and library for cell broadband engine processors. In *Proceeding (590) Parallel and Distributed Computing and Systems*. ACTA Press, 2007.

[26] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Cellss : a programming model for the cell be architecture. In *SC '06 : Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 86, New York, NY, USA, 2006. ACM.

[27] M. D. McCool. Data-parallel programming on the cell be and the gpu using the rapidmind development platform. 2006.

[28] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge : a programming model for heterogeneous multi-core systems. In *ASPLOS XIII*, 2008.

[29] M. Haldar, A. Nayak, A. Kanhere, P. Joisha, N. Shenoy, A. Choudhary, and P. Banerjee. A library-based compiler to execute matlab programs on a heterogeneous platform.

[30] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. Reiter Horn, L. Leem, J. Young Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia : Programming the memory hierarchy. In *Supercomputing*, 2006.

[31] B. Meister R. Lethin, A. Leung and E. Schweitz. R-stream : A parametric high level compiler. Technical report, Reservoir Labs, Inc.

[32] R. Dolbeau, S. Bihan, and F. Bodin. Hmpp : A hybrid multi-core parallel programming environment.

[33] John Stratton, Sam Stone, and Wen mei Hwu. Mcuda : An efficient implementation of cuda kernels on multi-cores. Technical Report IMPACT-08-01, University of Illinois at Urbana-Champaign, March 2008.

[34] Scott Pakin. Receiver-initiated Message Passing over RDMA Networks. In *IPDPS 2008*.

[35] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. Mpi microtask for programming the cell broadband enginetm processor. *IBM Syst. J.*, 45(1), 2006.

[36] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2) :3–35, 2001.

[37] Stéphanie Moreaud and Brice Goglin. Impact of NUMA Effects on High-Speed Networking with Multi-Opteron Machines. In *The 19th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2007)*, Cambridge, Massachussetts, November 2007.

[38] Stéphanie Moreaud. Impact des architectures multiprocesseurs sur les communications dans les grappes de calcul : de l'exploration des effets numa au placement automatique. Mémoire de dea, Université Bordeaux 1, June 2007.

[39] Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Building Portable Thread Schedulers for Hierarchical Multiprocessors : the BubbleSched Framework. In *EuroPar*, Rennes,France, 8 2007. ACM.

[40] Olivier Aumage, Elisabeth Brunet, Nathalie Furmento, and Raymond Namyst. Newmadeleine : a fast communication scheduling engine for high performance networks. In *CAC 2007 : Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2007*, Long Beach, California, USA, March 2007. Also available as LaBRI Report 1421-07 and INRIA RR-6085.

[41] J. Sancho and D. Kerbyson. Analysis of double buffering on two different multicore architectures : Quad-core opteron and the cell-be. *22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2008.

[42] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5) :720–748, 1999.

[43] Vincent Danjean, Raymond Namyst, and Pierre-André Wacrenier. An efficient multi-level trace toolkit for multi-threaded applications. In *EuroPar*, Lisbonne, Portugal, September 2005.

[44] Jean Garrigues. Initiation à la méthode des éléménts finis, 2002. http ://jgarrigues.perso.ec-marseille.fr/ef.html.

[45] Grégoire Allaire. *Analyse numérique et optimisation*. Éditions de l'École Polytechnique, 2005.

[46] Lawrence Agbezuge. Finite element solution of the poisson equation with dirichlet boundary conditions in a rectangular domain.